

DITTRICH
GELFAND
SCHEMME

Intern

HARDWARE	SCHNITTSTELLEN	EXEC	MULTITASKING	DOS
68000	Video	Mode	Priority	Functions
Agnus	Audio	List	Memory	I/O
Denise	Centronics	Library	Devices	Setup
Paula	Serial	Task	Resources	Task
CIA	Floppy	Exception	Task Switching	CIA
	Expansion	Trap	EXEC-base	Bus-Bus
	Keyboard	EXEC-base	Reset	

Ein **DATA BECKER** Buch

AMIGA

DITTRICH
GELFAND
SCHEMMEL

Intern

HARDWARE	SCIENTISTEN	EXEC	MULTITASKING	IOS
68000	Video	Tools	Priority	Features
Agona	Audio	Lib	Memory	IO
Device	Contronics	Library	Devices	Index
Panel	Serial	Task	Resources	Book
CIA	Floppy	Exception	Task-Switching	CIA
	Expansion	Trap		Book-Book
	Translator	EXEC-base		
		Reset		

Ein **DATA BECKER** Buch

AMIGA

ISBN 3-89011-104-1

Copyright © 1987 DATA BECKER GmbH
Merowingerstraße 30
4000 Düsseldorf

Alle Rechte vorbehalten. Kein Teil dieses Buches darf in irgendeiner Form (Druck, Fotokopie oder einem anderen Verfahren) ohne schriftliche Genehmigung der DATA BECKER GmbH reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

Wichtiger Hinweis:

Die in diesem Buch wiedergegebenen Schaltungen, Verfahren und Programme werden ohne Rücksicht auf die Patentlage mitgeteilt. Sie sind ausschließlich für Amateur- und Lehrzwecke bestimmt und dürfen nicht gewerblich genutzt werden.

Alle Schaltungen, technischen Angaben und Programme in diesem Buch wurden von dem Autoren mit größter Sorgfalt erarbeitet bzw. zusammengestellt und unter Einschaltung wirksamer Kontrollmaßnahmen reproduziert. Trotzdem sind Fehler nicht ganz auszuschließen. DATA BECKER sieht sich deshalb gezwungen, darauf hinzuweisen, daß weder eine Garantie noch die juristische Verantwortung oder irgendeine Haftung für Folgen, die auf fehlerhafte Angaben zurückgehen, übernommen werden kann. Für die Mitteilung eventueller Fehler ist der Autor jederzeit dankbar.

Inhaltsverzeichnis

1.	Die Hardware des Amiga.....	11
1.1	Einleitung.....	11
1.2	Die Komponenten des Amiga-Systems.....	12
1.2.1	Der 68000-Prozessor.....	13
1.2.2	Das CIA 8520	21
1.2.3	Die Customchips und ihre Einbindung in die Amiga-Hardware.....	39
1.2.3.1	Der Grundaufbau des Amiga	41
1.2.3.1	Der Aufbau von Agnus.....	46
1.2.3.2	Der Aufbau von Denise	52
1.2.3.3	Der Aufbau von Paula	57
1.2.3.4	Besonderheiten des A500	61
1.3	Die Schnittstellen des Amiga.....	65
1.3.1	Die Audio/Video-Schnittstellen	66
1.3.2	Die RGB-Buchse.....	69
1.3.3	Die Centronics-Schnittstelle	72
1.3.4	Die serielle Schnittstelle	75
1.3.5	Die Anschlußbuchse für externe Disketten- laufwerke	78
1.3.6	Die Gameports.....	87
1.3.7	Der Expansion-Port.....	90
1.3.8	Stromversorgung über die Schnittstellen.....	94
1.4	Die Tastatur	96
1.4.1	Die Schaltung der Tastaturplatine	98
1.4.2	Die Datenübertragung	100
1.4.3	Schwächen der Tastatur	104
1.5	Die Programmierung der Hardware.....	106
1.5.1	Die Speicherbelegung	107
1.5.2	Grundlagen	119
1.5.3	Interrupts.....	137
1.5.4	Der Coprozessor Copper	140
1.5.5	Playfields.....	153
1.5.6	Sprites.....	189
1.5.7	Der Blitter	208
1.5.8	Die Tonausgabe	257
1.5.9	Maus, Joystick und Paddles.....	294

1.5.10	Die serielle Schnittstelle.....	304
1.5.11	Der Diskcontroller.....	311
2.	Exec	317
2.1	Grundlagen des Betriebssystems.....	317
2.2	Einführung in die Programmierung des Amiga.....	318
2.2.1	Unterschiede bei C und Assembler.....	318
2.2.2	Aufbau von Nodes (Knoten).....	321
2.2.3	Listen.....	325
2.2.4	Exec-Routine zur Listenverwaltung	328
2.3	Funktionstabellen (Libraries).....	335
2.3.1	Öffnen einer Library	341
2.3.2	Schließen einer Library.....	343
2.3.3	Struktur der Library.....	344
2.3.4	Ändern einer bestehenden Library	347
2.3.5	Das Erstellen einer eigenen Library	347
2.3.6	Beschreibung der restlichen Library- Funktionen	356
2.4	Multitasking	357
2.4.1	Die Task-Struktur	358
2.4.1.1	Task-Funktionen	371
2.4.2	Kommunikation zwischen Tasks	374
2.4.2.1	Die Task-Signale	375
2.4.2.2	Das Message-System.....	381
2.5	Speicherverwaltung des Amiga.....	401
2.5.1	Die Funktionen AllocMem() und FreeMem()	404
2.5.2	Die Memory-List-Struktur	406
2.5.3	Speicherzuweisung und Tasks.....	410
2.5.4	Die interne Verwaltung des Speichers	411
2.5.5	Die Allocate- und Deallocate-Funktion	414
2.5.6	Beschreibung der restlichen Funktionen.....	416
2.6	IO-Handhabung auf dem Amiga	417
2.6.1	Aufbau der IOREquest-Struktur	417
2.6.2	Aufbau eines Devices.....	420
2.6.3	IO-Steuerung über Funktionen	424
2.7	Interrupt-Handhabung auf dem Amiga.....	430
2.7.1	Aufbau der Interrupt-Strukturen	432
2.7.2	Soft-Interrupts	440

2.7.3	Die CIA-Interrupts.....	443
2.7.3.1	Die CIA-Resource-Struktur	444
2.7.3.2	Die Verwaltung der Resource-Struktur	447
2.7.4	Beschreibung der Interrupt-Funktionen	453
2.7.5	Beispiel eines Interrupt-Servers	455
2.8	Die ExecBase-Struktur.....	458
2.9	Reset-Routine und resetfeste Programme	470
2.9.1	Dokumentation der Reset-Routine	470
2.9.2	Resident-Strukturen	480
2.9.3	Resetfeste Programme und Strukturen	487
2.9.4	Ein richtiges NoFastMem	492
3.	Das AmigaDOS.....	495
3.1	Die DOS-Bibliothek	495
3.1.1	Laden der DOS.LIBRARY	495
3.1.2	Funktionsaufruf und Parameterübergabe	497
3.1.3	Die DOS-Funktionen	498
3.1.4	DOS-Fehlermeldungen	516
3.2	Disketten	520
3.2.1	Der Boot-Vorgang.....	522
3.2.2	Datei-Strukturen und Daten-Verteilung.....	523
3.2.2.1	Diskettenaufteilung.....	524
3.2.2.2	Programmaufbau.....	532
3.2.2.3	Das IFF-Format.....	540
3.3	Programme	547
3.3.1	Programmstart und Parameter	548
3.3.1.1	Aufruf mit CLI	548
3.3.1.2	Start von der Workbench aus.....	551
3.3.2	Aufbau der transienten CLI-Befehle	560
3.4	Ein-/Ausgabe	564
3.4.1	Standard-I/O.....	565
3.4.1.1	Tastatur und Bildschirm.....	567
3.4.1.2	Disketten-Dateien.....	575
3.4.1.3	Serielle Schnittstelle.....	577
3.4.1.4	Parallele Schnittstelle.....	578
4.	Devices	579
4.1	Trackdisk-Device: Zugriff auf Disketten.....	581
4.2	Consol-Device: Editor-Fenster.....	595
4.3	Narrator-Device: Sprachausgabe	602

4.4	Serial-Device: die RS232-Schnittstelle.....	606
4.5	Printer-Device: Drucker-Programmierung	610
4.6	Parallel-Device: Digital-Ein-/Ausgaben.....	612
4.7	Gameport-Device: Maus und Joystick.....	613

Anhang

A:	Bibliotheksfunktionen im Überblick	619
B:	Stichwortverzeichnis.....	630

1. Die Hardware des Amiga

1.1 Einleitung

Der Amiga von Commodore bietet dem Anwender Möglichkeiten, von denen noch vor einigen Jahren bei Computern dieser Preisklasse niemand zu träumen gewagt hätte. Um diese Leistungen zu ermöglichen, arbeiten beim Amiga ein leistungsfähiges Betriebssystem und eine ausgeklügelte Hardware eng zusammen.

Ein Ziel der Entwickler dieses Computers war eine hohe Benutzerfreundlichkeit. Man wollte nach dem Vorbild eines Apple Macintosh mit Hilfe der Maus und der Workbench als grafischer Benutzeroberfläche dem Anwender die Bedienung seines Computers erleichtern. Aber nicht nur dieser sollte leicht mit dem Gerät zurechtkommen, auch der Programmierer wurde entlastet. Für beinahe jede erdenkliche Aufgabe findet man im Betriebssystem eine passende Routine, die die direkte Programmierung der Hardware anscheinend überflüssig macht.

Aber eben doch nur anscheinend. Denn trotz all dieser komfortablen Routinen kommt man um die direkte Programmierung nicht immer herum, denn die Geschwindigkeit der Betriebssystemroutinen ist sehr viel niedriger, als man es vom Amiga erwarten würde. Der Grund dafür ist die Programmiersprache C, in der das Amiga-Betriebssystem größtenteils geschrieben wurde.

Will man also schnelle und leistungsfähige Programme schreiben oder will man ganz einfach nur seinen Amiga besser kennenlernen, sollte man sich mit der Hardware beschäftigen. Das folgende Kapitel bietet dazu eine Beschreibung der Amiga-Hardware und der Programmierung der einzelnen Chips.

1.2 Die Komponenten des Amiga-Systems

Im wesentlichen besteht die Hardware des Amiga aus folgenden Bausteinen, egal ob es sich um einen A500, 1000 oder 2000 handelt:

- Der 68000er von Motorola als Mikroprozessor
- Zwei Schnittstellenbausteine vom Typ 8250
- Drei Spezialchips von Commodore, genannt Agnus, Denise und Paula

Läßt man das RAM und die unumgänglichen Logikbausteine einmal außer acht, sind die sechs oben genannten Chips für alle Funktionen des Amiga verantwortlich.

Auch an Schnittstellen ist beim Amiga alles Nötige vorhanden:

- Paralleler Druckerport (Centronics)
- Serielle RS232-Schnittstelle
- RGB-Monitoranschluß
- Composite Video (nicht A2000)
- Stereo-Audioausgang
- Anschlußbuchse für einen HF-Modulator
- Commodore-eigener Tastaturanschluß
- Anschluß für externe Floppylaufwerke (Shugart-Bus-kompatibel)
- Zwei identische Buchsen zum Anschluß verschiedener Eingabegeräte wie Maus, Joystick und Paddle

- Anschluß für 256 KByte RAM-Erweiterung (Auf diesen Anschluß wird in diesem Buch nicht weiter eingegangen, da hier nur die Original RAM-Erweiterung zur Erweiterung von 256 auf 512 KByte angeschlossen werden kann. Beim A500 und A2000 sind diese 256 KByte schon von Anfang an eingebaut. Der A500 hat dafür einen Anschluß für eine RAM-Erweiterung um 512 KByte, der aber völlig anders geartet ist.)
- Expansionport zum Anschluß von Systemerweiterungen aller Art (Dieser Anschluß liegt beim A500 und A1000 an der Gehäusesseite, beim A2000 ist er in Form mehrerer Steckplätze ins Geräteinnere verlegt worden.)

Um die Zusammenarbeit aller Bausteine im Amiga zu verstehen, muß erst einmal die Funktion der einzelnen Chips geklärt werden.

1.2.1 Der 68000-Prozessor

Der 68000 von Motorola ist unumstritten einer der leistungsfähigsten 16-Bit-Prozessoren. Obwohl er schon seit 1979 auf dem Markt ist, findet man ihn erst seit kurzem in Computern der Preisklasse eines Amiga.

Natürlich kann hier keine ausführliche Beschreibung des 68000 erwartet werden, denn dies würde den Rahmen dieses Buches sprengen. Wer mehr über die Programmierung des 68000 wissen möchte, sei an die entsprechende Fachliteratur verwiesen. An dieser Stelle soll lediglich die Pinbelegung und eine kurze Beschreibung der einzelnen Signalgruppen stehen, da viele Bücher über die Programmierung des 68000 zwar eine gute Einführung in die Softwareseite bieten, aber kaum etwas über die Hardware sagen. Eine grundlegende Kenntnis der Signale des 68000 ist aber für das Verständnis der Amiga-Hardware unumgänglich.

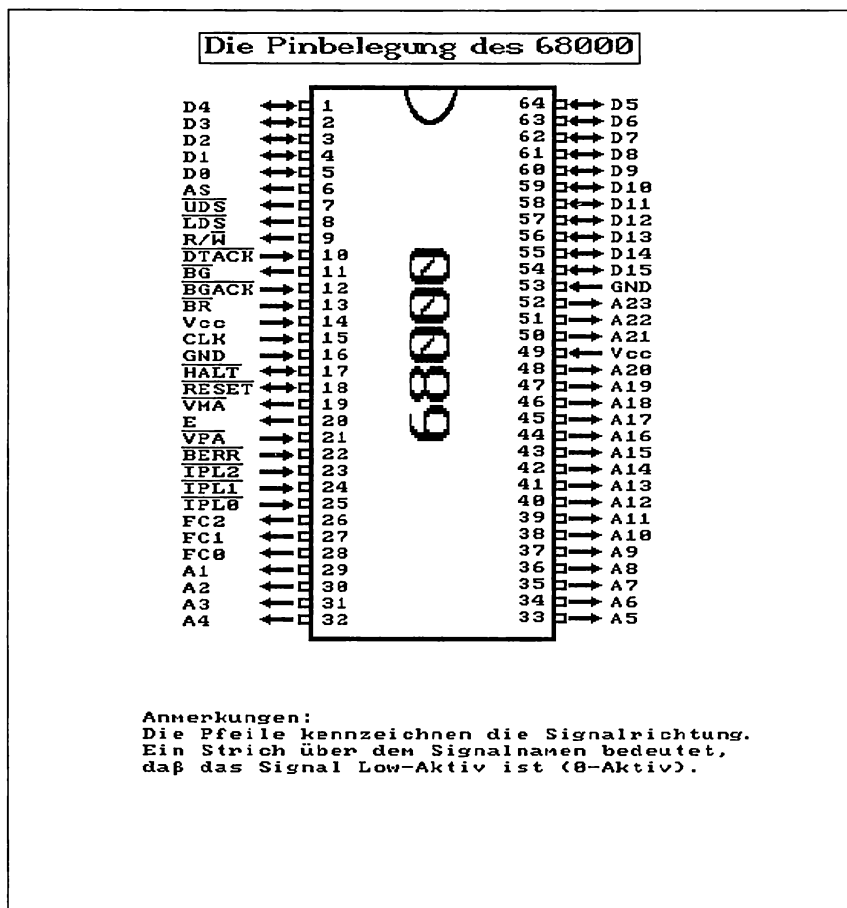


Abb. 1.2.1.1

Man kann die Anschlüsse in folgende Funktionsgruppen aufteilen:

Die Stromversorgung: VCC und GND

Der 68000 arbeitet mit einer einfachen Versorgungsspannung von 5 Volt. Die Anschlüsse sind doppelt ausgelegt und zentral gelegen, um durch kurze Leitungswege im Gehäuse die Stromverluste minimal zu halten.

Der Takteingang: CLK

Der 68000 benötigt lediglich einen einfachen Takt. Die Frequenz hängt von der gewählten Prozessorversion ab. Im Amiga beträgt die Taktfrequenz für den Prozessor 7.16 MHz.

Der Datenbus: D0 - D15

Der Datenbus ist als 16 Bit Bus ausgelegt und kann somit ein Wort (16 Bit) auf einmal übertragen. Beim Transfer einzelner Bytes (8 Bit) ist jeweils nur eine Hälfte der Leitungen an der Datenübertragung beteiligt. Entweder wird das Byte über die unteren 8 Bit oder über die oberen 8 Bit gelesen bzw. geschrieben.

Der Adreßbus: A1 - A23

Der Adreßbus kann mit seinen 23 Leitungen 8 Megaworte Speicher ansprechen (2^{23} entspricht 8 Megaworten oder 16 Megabytes). Da er kein A0 Adreßbit besitzt, kann er diesen Speicher nur wortweise adressieren.

Bussteuerleitungen im asynchronen Modus: AS, R/W, UDS, LDS, DTACK

Der 68000 kann seine Speicherzugriffe grundsätzlich in zwei verschiedenen Modi ausführen. Im asynchronen Modus signalisiert der Prozessor mit AS (Adress-Strobe/Adresse gültig), daß eine gültige Adresse am Adreßbus anliegt. Gleichzeitig bestimmt er mit R/W (Read-Write/Lesen-Schreiben), ob ein Byte/Wort gelesen oder geschrieben werden soll. Die Wahl zwischen Wort oder Byte treffen die beiden Leitungen UDS und LDS (Upper-, Lower Data Strobe/Obere, Untere Datenbushälfte). Da der Speicher immer wortweise adressiert wird, überträgt der Pro-

zessor bei einem Bytezugriff einfach nur die oberen oder die unteren 8 Bits des Datenbusses. Dies signalisiert er durch UDS und LDS. Bei einem Wortzugriff legt der 68000 beide Leitungen auf 0. Will er dagegen auf ein Byte zugreifen, legt er entweder UDS oder ODS auf 0, die andere Leitung bleibt 1.

Hat der Prozessor jetzt mit AS, R/W, UDS und LDS den von ihm gewünschten Zugriff signalisiert, wartet er, bis der Speicher ihm mitteilt, daß die gewünschten Daten bereit sind. Dazu verwendet dieser die Leitung DTACK, die er auf 0 legt, sobald er die Daten bereitgestellt hat. Schreibt der Prozessor Daten, teilt ihm der Speicher durch DTACK mit, daß er die Daten übernommen hat.

Im asynchronen Modus paßt sich der Prozessor also immer an die Geschwindigkeit des Speichers an.

Die einzelnen Worte und Bytes liegen dann folgendermaßen im Speicher:

Beteiligte Datenbusleitungen:			
		D8-15	D0-7
Adresse:		UDS=0	LDS=0
0	Wort 0	Byte 0	Byte 1
2	Wort 1	Byte 2	Byte 3
4	Wort 2	Byte 4	Byte 5
6	Wort 3	Byte 6	Byte 7
...	...	...	...

Bussteuersignale im synchronen Modus: E, VPA, VMA

Um den Sinn dieser Signale besser verstehen zu können, muß man die Situation zum Zeitpunkt der Markteinführung des 68000 kennen. Es waren damals keine eigenen Peripheriechips für den 68000 verfügbar. Die vorhandenen Chips von Motorola, die ja für die 6800-Serie (von der auch der 6502 abstammt) gedacht waren, konnten nicht ohne zusätzliche Schaltungen in die asynchrone Bussteuerung eingepaßt werden. Also versah man den 68000 bei Motorola noch mit einem synchronen Busmodus,

wie man ihn von den 8-Bit-Prozessoren wie dem 6800 oder 6502 kennt.

An der Leitung E liegt dabei ständig ein durch den Faktor zehn geteilter Prozessortakt an, beim Amiga also 716 KHz, der den Peripheriechips als Takt dient. (Er wird mit dem Phi-2-Eingang der Peripheriechips verbunden.) Die Umschaltung von dem synchronen Modus in den asynchronen erfolgt über den Eingang VPA (Valid Periphial Adress / Gültige Peripherieadresse). Dieser Eingang muß von einem externen Adreßdecoder auf 0 gelegt werden, sobald dieser die Adresse eines Peripheriechips erkennt. Der Prozessor antwortet darauf, indem er die Leitung VMA (Valid Memory Adress / Gültige Speicheradresse) ebenfalls auf 0 legt. Das entsprechende Peripheriechip muß jetzt innerhalb eines Taktzyklus von E die Daten übernehmen bzw. bereitstellen. Danach verläßt der 68000 automatisch den synchronen Modus, bis das VPA Signal erneut aktiv wird.

Dies bedeutet also, daß ein Peripheriechip im synchronen Modus gezwungen ist, die Daten an einem bestimmten Zeitpunkt bereitzustellen bzw. zu übernehmen.

Die Steuersignale des Systems: RESET, HALT, BERR

Die wichtigste Aufgabe eines Reset-Signals ist das Zurücksetzen des Systems, so daß alle Systemkomponenten in einen verlässlichen Grundzustand versetzt werden und die weitere Programmausführung an einer festgelegten Adresse beginnt.

Um einen solchen Systemreset auszulösen, muß beim 68000 sowohl die HALT- als auch die RESET-Leitung auf 0 gelegt werden. Sobald diese Leitungen dann wieder auf 1 gehen, beginnt der 68000 mit der Programmausführung bei der Adresse, die er in der Speicheradresse 4 vorfindet.

Die Leitung RESET kann auch vom 68000 aus auf 0 gelegt werden, um das System zu initialisieren, ohne den Prozessorzustand zu verändern.

Mit der BERR-Leitung (Bus-Error/Bus-Fehler) kann eine externe Überwachungsschaltung dem Prozessor mitteilen, daß irgend etwas nicht in Ordnung ist. Ein Grund für einen Bus-Fehler kann z.B. ein Hardware-Defekt oder ein Zugriff des Prozessors auf eine nicht existierende Adresse sein.

Tritt ein BERR-Signal auf, springt der 68000 in eine spezielle Routine des Betriebssystems, die dann die weitere Fehlerbehandlung übernimmt (Guru-Meditation läßt grüßen!). Tritt während dieser Fehlerbehandlung ein erneuter Bus-Fehler auf, hält der 68000 jede Programmausführung an und legt HALT auf 0. Dieser sogenannte doppelte Bus-Fehler ist übrigens der einzige Fall, in dem der 68000 abstürzt, d.h. jegliche Programmausführung einstellt. Bei sämtlichen anderen Fehlern springt er über spezielle Vektoren in Programmroutinen, die dann die Fehlerbehandlung übernehmen können und eine Weiterarbeit des Systems ermöglichen. Beim Amiga hat man hier im Betriebssystem etwas gespart. (Man beachte die Häufigkeit der Guru-Meditationen, die sich beim Amiga getreu Murphy's Gesetz verhalten: Ein Computer stürzt immer genau dann ab, wenn man wichtige Daten bearbeitet, die man noch nicht gespeichert hat.)

Hat der Prozessor die Programmausführung auf Grund eines doppelten Bus-Fehlers angehalten, kann er nur mittels einem Reset wieder gestartet werden (HALT und RESET auf 0).

Eine weitere Funktion der HALT-Leitung ist ein Stopp des Prozessors. Legt man HALT auf 0, beendet der 68000 noch den aktuellen Speicherzugriff und wartet dann, bis HALT wieder auf 1 zurückgesetzt wird.

Es gibt noch einige weitere Details im Zusammenspiel von BERR und HALT, die hier nicht weiter erwähnt werden sollen, da sie im Amiga ohne Belang sind.

Der Betriebszustand des Prozessors: FC0, FC1, FC2

Die Leitungen FC0 - FC2 signalisieren den Betriebszustand des Prozessors. Folgende Zustände sind möglich:

FC2	FC1	FC0	Zustand:
0	0	1	Zugriff auf Anwender-Daten
0	1	0	Zugriff auf Anwender-Programm
1	0	1	Zugriff auf Supervisor-Daten
1	1	0	Zugriff auf Supervisor-Programm
1	1	1	Signalisierung eines gültigen Interrupts

Der Prozessor kann grundsätzlich in zwei verschiedenen Modi betrieben werden, nämlich im Anwender-Modus und im Supervisor-Modus (Überwacher-Modus). Ein Programm, das im Supervisor-Modus läuft, hat uneingeschränkten Zugriff auf sämtliche Prozessorregister. Das Betriebssystem arbeitet z.B. immer im Supervisor-Modus.

Im Anwender-Modus sind bestimmte Register des Prozessors für das Programm gesperrt. Genauereres darüber finden Sie in der Fachliteratur zum 68000.

Die drei FCx Leitungen erlauben es also der Systemhardware, den aktuellen Zustand des Prozessors zu erkennen und möglicherweise darauf zu reagieren. So kann zum Beispiel im Anwender-Modus beim Zugriff auf Speicherbereiche des Betriebssystems ein Bus-Fehler (BERR = 0) erzeugt werden.

Die Unterbrechungseingänge (Interrupteingänge): IPL0, IPL1, IPL2

Die Signale an den drei Interrupteingängen (IPL = Interrupt Pending Level) werden vom 68000 als eine 3 Bit Binärzahl interpretiert. Der 68000 kann also acht verschiedene Interrupt-Signale, die sogenannten Interrupt-Ebenen, unterscheiden, wobei eine 0 bedeutet, daß kein Interrupt anliegt, während eine 7 einen Interrupt höchster Priorität signalisiert. Jeder der sieben Interrupt-Ebenen ist ein eigener Interrupt-Vektor zugewiesen, der die Adresse der Routine enthält, in die bei einem Interrupt verzweigt wird.

Ist ein Interrupt der entsprechenden Ebene erlaubt, legt der Prozessor alle FCx-Leitungen auf 1 und signalisiert damit, daß er einen Interrupt erkannt hat und jetzt auf eine Bestätigung von Seiten des Verursachers wartet. Sie kann durch VPA oder DTACK erfolgen. Bei einer Bestätigung durch VPA erfolgt eine sogenannte autovektorielle Unterbrechung, d.h. der Prozessor springt an die Adresse, die er in dem der Interrupt-Ebene zugewiesenen Vektor findet. D.h. es können sieben verschiedene Adressen angesprungen werden, da es sieben gültige Interrupt-Ebenen gibt (Ebene 0 bedeutet ja, daß kein Interrupt vorliegt).

Gibt es nur sieben verschiedene Interrupt-Quellen in dem System, braucht also keine softwaremäßige Trennung der einzelnen Quellen mehr vorgenommen werden. Man ordnet einfach jeder Interrupt-Ebene eine Interrupt-Quelle zu, und der Prozessor springt dann in das entsprechende Programm. Der Amiga verwendet nur diese autovektorielle Unterbrechungen.

Noch mehr Möglichkeiten zur hardwaremäßigen Trennung von verschiedenen Interrupt-Quellen bietet die sogenannte nicht-autovektorielle Unterbrechung. Da sie im Amiga nicht verwendet wird, soll auch nicht weiter darauf eingegangen werden. Es sei nur soviel gesagt, daß bei einer nicht-autovektoriellen Unterbrechung die Bestätigung durch DTACK erfolgen muß und der Baustein, der den Interrupt ausgelöst hat, einen Interrupt-Vektor auf den Datenbus legen kann, der dann eine hardwaremäßige Unterscheidung von bis zu 192 verschiedenen Interrupt-Vektoren erlaubt.

Signale für die Buszuweisung: BR, BG, BGACK

Diese drei Signale erlauben es einem anderen Chip den Bus zu übernehmen. Dies kann zum Beispiel bei einem Harddisk-Controller der Fall sein, der dann die Daten von der Festplatte direkt in den Speicher schreibt (sogenannter DMA = Direct Memory Access / Direkter Speicherzugriff).

Auch diese Signale sind im Amiga nicht benutzt, hier wird der DMA auf eine andere Weise realisiert, die am Ende dieses Kapitels noch näher beschrieben wird.

1.2.2 Das CIA 8520

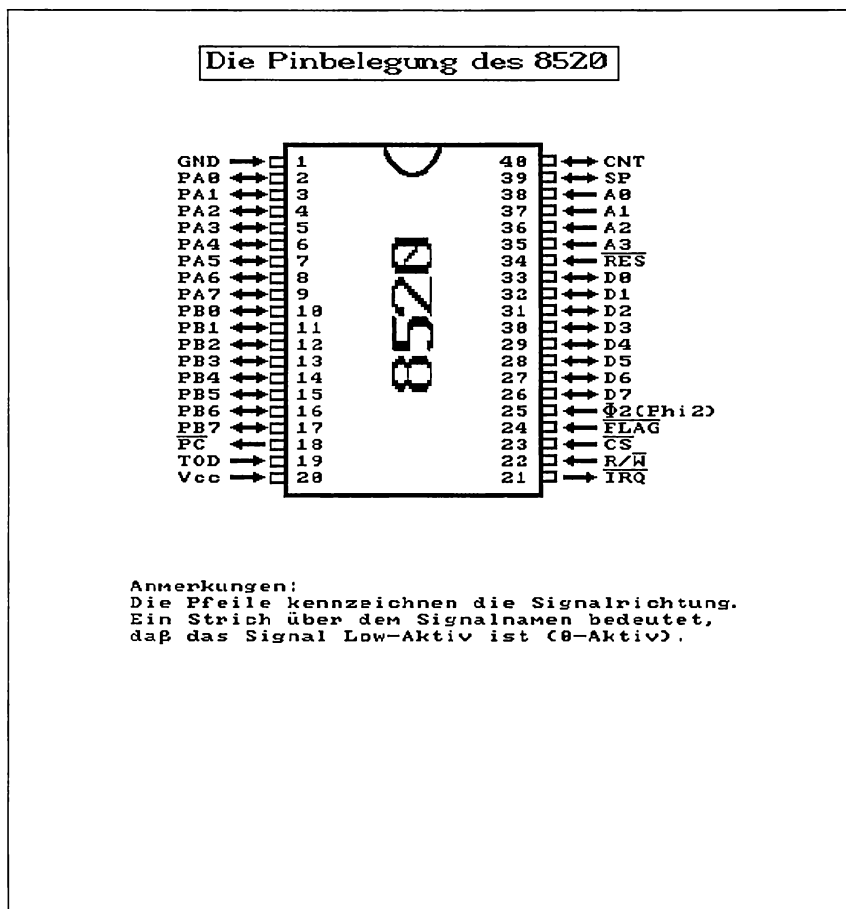


Abb. 1.2.2.1

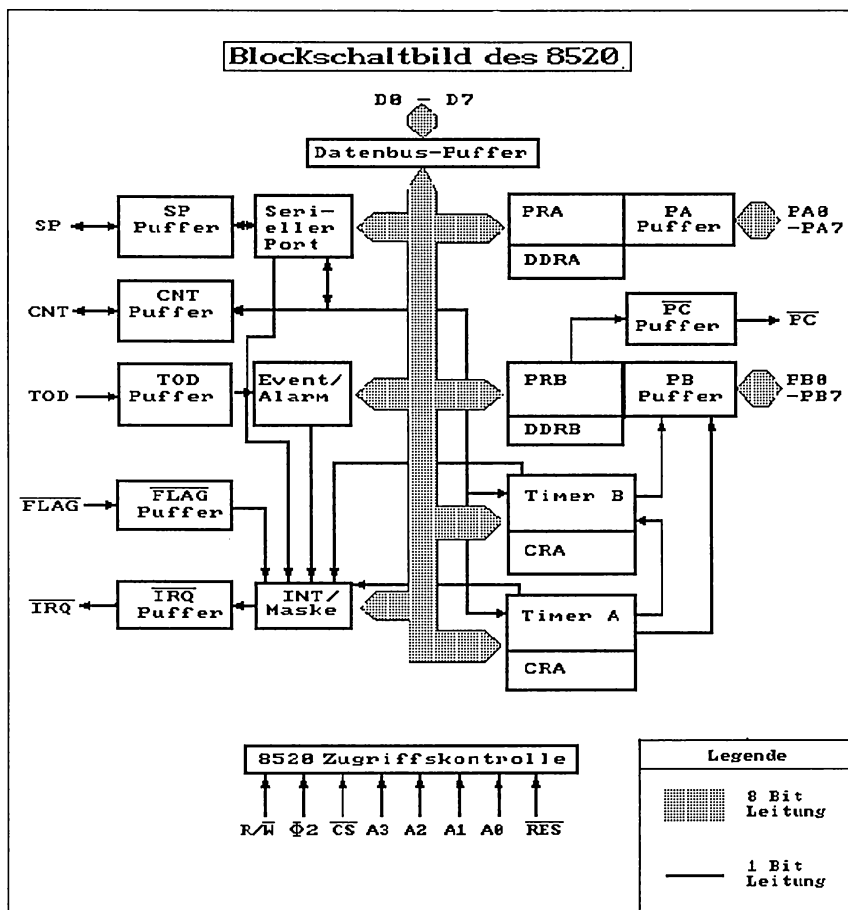


Abb. 1.2.2.2

Der 8520 ist ein Peripheriebaustein vom Typ eines sogenannten Complex Interface Adapters (CIA), was soviel wie vielseitiger Schnittstellenbaustein bedeutet. Dies heißt, daß die Entwickler des 8520 versucht haben, möglichst viele Funktionen in einem Chip unterzubringen. Betrachtet man den 8520 genauer, fällt einem sofort eine große Ähnlichkeit mit einem alten Bekannten auf. Der 8520 gleicht nämlich dem 6526, der im C64 seine Dienste versah, fast wie ein Ei dem anderen. Lediglich die

Funktionsweise der Register 8 bis 11 (\$8 bis \$B) hat sich etwas verändert. Dies ist sicher eine erfreuliche Mitteilung für alle, die sich in der Programmierung des 6526 auskennen.

Der 8520 verfügt im einzelnen über folgende Möglichkeiten: zwei frei programmierbare 8-Bit-Parallelports (PA und PB), zwei 16-Bit-Abwärtszähler (Timer A und Timer B), einen bidirektionalen seriellen Port (SP) und einen 24-Bit-Zähler (Event Counter) mit Alarmfunktion beim Erreichen eines vorgewählten Wertes. Sämtliche Funktionen sind in der Lage, Interrupts auszulösen.

Organisiert sind die Funktionen des 8520 in 16 Registern. Für den Prozessor erscheinen sie als ganz normale Speicherstellen, da sämtliche Peripheriebausteine in einem 68000-System, und damit auch im Amiga, memory mapped sind, d.h. die Register dieser Chips erscheinen als Speicherstellen, die mit den üblichen Befehlen wie z.B. Move gelesen und beschrieben werden können.

Da der 8520 vom 6526 abstammt, der ja für die 8-Bit-Prozessoren der 65xx Reihe entwickelt wurde, muß der 68000 ihn im synchronen Modus ansprechen (siehe Kapitel 1.2.1).

Der E-Takt des 68000 ist deshalb auch mit dem Phi2-Eingang des 8520 verbunden. Die Auswahl der 16 internen Register geschieht mit den vier Adreßeingängen des 8520: A0 - A3. Genaue Details über die Einbindung der CIAs in das Amiga-System finden sich am Ende dieses Kapitels.

Hier erst einmal die Belegung der 16 Register (Eigentlich sind es nur 15 Register, da das Register 11 (\$B) unbenutzt ist):

Registerbelegung des 8520

Register	Name	Funktion
0 0	PRA	Datenregister für Port A
1 1	PRB	Datenregister für Port B
2 2	DDRA	Datenrichtungsregister für Port A
3 3	DDRB	Datenrichtungsregister für Port B
4 4	TALO	Timer A untere 8 Bit
5 5	TAHI	Timer A obere 8 Bit
6 6	TBLO	Timer B untere 8 Bit
7 7	TBHI	Timer B obere 8 Bit
8 8	Event Lo	Zähler Bits 0-7
9 9	E. 8-15	Zähler Bits 8-15
10 A	Event Hi	Zähler Bits 16-23
11 B	---	unbenutzt
12 C	SP	Datenregister des seriellen Ports
13 D	ICR	Interrupt Kontroll Register
14 E	CRA	Kontrollregister A
15 F	CRB	Kontrollregister B

Die Parallelports

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
0	PRA	PA7	PA6	PA5	PA4	PA3	PA2	PA1	PA0
1	PRB	PB7	PB6	PB5	PB4	PB3	PB2	PB1	PB0
2	DDRA	DPA7	DPA6	DPA5	DPA4	DPA3	DPA2	DPA1	DPA0
3	DDRB	DPB7	DPB6	DPB5	DPB4	DPB3	DPB2	DPB1	DPB0

Der 8520 verfügt über zwei 8-Bit-Parallelports, PA und PB, denen jeweils ein Datenregister zugeordnet ist, PRA und PRB (Port Register). Dementsprechend verfügt das Chip über 16 Portleitungen, PA0 - PA7 und PB0 - PB7. Jede Portleitung kann sowohl als Ein- als auch als Ausgang verwendet werden. Dies wird als Datenrichtung bezeichnet. Der 8520 erlaubt es, die Datenrichtung jeder Leitung getrennt einzustellen. Dazu besitzt jeder Port ein korrespondierendes Datenrichtungsregister, DDRA und DDRB (Data Direction Register). Ist ein Bit im Datenrichtungsregister 0, ist die entsprechende Leitung als Eingang

geschaltet. Ihr Zustand kann durch Lesen des entsprechenden Bits des Datenregisters abgefragt werden.

Setzt man ein Bit im Datenrichtungsregister auf 1, schaltet man die entsprechende Leitung als Ausgang. Der Pegel an der zugehörigen Portleitung stellt dann direkt den Wert des korrespondierenden Bits des Datenregisters dar.

Im allgemeinen gilt, daß beim Schreiben in das Datenregister der Wert dort immer gespeichert wird, während beim Lesen immer der Zustand der Portleitungen auf dem Datenbus erscheint. Die Bits im Datenrichtungsregister bestimmen, ob der Wert des Datenregisters auf die Portleitungen geschaltet wird. Deshalb erhält man beim Lesen eines Ports, der als Ausgang geschaltet ist, den Inhalt des Datenregisters, während beim Schreiben in einen Eingangsport der Wert im Datenregister gespeichert wird, aber erst auf den Portleitungen erscheint, wenn man auf Ausgabe umschaltet.

Um die Datenübertragung mittels der Parallelports zu vereinfachen, besitzt jeder 8520 zwei Handshake-Leitungen, PC und FLAG.

Der PC-Ausgang geht bei jedem Zugriff auf das Datenregister B (PRB, Reg. 1) für einen Taktzyklus auf 0. Der Eingang FLAG hingegen reagiert auf solche negativen Flanken. Jedesmal wenn der Zustand an der FLAG-Leitung von 1 auf 0 wechselt, wird im Interrupt-Kontrollregister (ICR, Reg. \$D) das FLAG-Bit gesetzt. Mit diesen beiden Leitungen kann somit auf einfache Weise ein Handshake realisiert werden, indem man die FLAG- und PC-Leitungen zweier CIAs wechselseitig miteinander verbindet.

Der Sender braucht lediglich seine Daten in das Portregister zu schreiben und vor jedem weiteren Byte auf ein FLAG-Signal warten. Da FLAG einen Interrupt auslösen kann, hat der Sender sogar die Möglichkeit, sich in den Wartepausen anderen Aufgaben zuzuwenden. Für den Empfänger gilt genau dasselbe, nur liest er die Daten vom Port, statt sie auszugeben.

Die Abwärtszähler/Timer:

Lesezugriff (Read)

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
4	TALO	TAL7	TAL6	TAL5	TAL4	TAL3	TAL2	TAL1	TAL0
5	TAHI	TAH7	TAH6	TAH5	TAH4	TAH3	TAH2	TAH1	TAH0
6	TBLO	TBL7	TBL6	TBL5	TBL4	TBL3	TBL2	TBL1	TBL0
7	TBHI	TBH7	TBH6	TBH5	TBH4	TBH3	TBH2	TBH1	TBH0

Schreibzugriff (Write)

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
4	PALO	PAL7	PAL6	PAL5	PAL4	PAL3	PAL2	PAL1	PAL0
5	PAHI	PAH7	PAH6	PAH5	PAH4	PAH3	PAH2	PAH1	PAH0
6	PBLO	PBL7	PBL6	PBL5	PBL4	PBL3	PBL2	PBL1	PBL0
7	PBHI	PBH7	PBH6	PBH5	PBH4	PBH3	PBH2	PBH1	PBH0

Der 8520 verfügt über zwei 16-Bit-Abwärtszähler (Timer). Diese Timer sind in der Lage, von einem voreingestellten Wert aus bis auf Null herunterzuzählen. Dabei sind eine Vielzahl verschiedener Modi möglich, die mit Hilfe eines Kontrollregisters gewählt werden können. Es existiert dabei für jeden Timer ein eigenes Kontrollregister (CRA und CRB).

Jeder Timer besteht intern aus vier Registern (für Timer A: TALO+TAHI und PALO+PAHI) oder zwei Registerpaaren, da je ein Low- und High-Register zusammen den 16-Bit-Zählerwert bilden. Beide Registerpaare liegen an der selben Adresse, aber das eine kann nur gelesen und das andere nur beschrieben werden. Bei einem Schreibzugriff auf eines der Timer-Register wird der Wert in einem Latch gespeichert. Dieser Wert wird in das Zählregister geladen und heruntergezählt, bis der Zähler einen Unterlauf, d.h. Null, erreicht hat. Danach wird der Wert aus dem Latch erneut in das Zählregister geladen.

Liest man eines der Timerregister, erhält man den aktuellen Stand des Zählregisters. Um hierbei einen korrekten Wert zu er-

halten, muß allerdings der Zähler angehalten werden. Warum, zeigt folgendes Beispiel:

Zählerstand: \$0100 Ein Lesezugriff auf das Register 5 ergibt das Highbyte des aktuellen Zählerstands: \$01

Bevor das Lowbyte (Reg. 4) gelesen werden kann, taucht ein Zählimpuls auf und erniedrigt den Zähler: Zählerstand jetzt \$00FF

Das Lowbyte wird gelesen: \$FF

Gelesener Zählerstand also: \$01FF!

Statt den Zähler anzuhalten, was ja auch Fehler verursacht, da jetzt Zählimpulse übergangen werden, kann man auch folgende elegantere Methode anwenden: Man liest das Highbyte, dann das Lowbyte, anschließend noch einmal das Highbyte. Ergibt der Vergleich beider gelesener Highbytes Übereinstimmung, ist der gelesene Wert korrekt. Fällt der Vergleich dagegen negativ aus, muß der Vorgang wiederholt werden.

Welche Signale den Zähler erniedrigen, bestimmen für Timer A Bit 5 und für Timer B die Bits 5 und 6 des jeweiligen Kontrollregisters.

Bei Timer A sind nur zwei Quellen möglich:

1. Timer A wird mit jedem Taktzyklus erniedrigt (da die CIAs im Amiga am E-Takt des Prozessors hängen, beträgt die Zählfrequenz 716 Khz) (INMODE = 0)
2. Jeder High-Impuls auf der CNT-Leitung erniedrigt den Zähler. (INMODE = 1)

Für Timer B gibt es vier Eingangsmodi:

1. Taktzyklen (INMODE-Bits = 00 (Binärdarstellung, erste Ziffer steht für Bit 6, die zweite für Bit 5.))
2. CNT-Impulse (INMODE-Bits = 01)

3. Unterläufe von Timer A (Damit kann man beide Timer zu einem 32 Bit Timer kombinieren.) (INMODE-Bits = 10)
4. Unterläufe von Timer A wenn die CNT-Leitung gleichzeitig High ist (Damit kann man auf einfache Weise die Länge eines Impulses an der CNT-Leitung messen.) (INMODE-Bits = 11)

Die Unterläufe eines Zählers werden im Interrupt-Kontrollregister (ICR) registriert. Bei einem Unterlauf von Timer A wird das TA-Bit (Bit Nr. 0) gesetzt, bei Timer B entsprechend TB (Bit Nr. 1). Diese Bits bleiben, wie alle Bits im ICR, gesetzt, bis das ICR gelesen wird. Zusätzlich besteht die Möglichkeit, die Unterläufe der Timer auf dem Parallelport B auszugeben. Wird das PBon Bit im Kontrollregister des jeweiligen Zählers (CRA oder CRB) gesetzt, erscheint jeder Unterlauf auf der entsprechenden Portleitung (PB 6 für Timer A und PB 7 für Timer B).

Mit dem OUTMODE-Bit kann man zwischen zwei verschiedenen Ausgabearten wählen:

- | | | |
|-------------|-------------|--|
| OUTMODE = 0 | Pulse-Mode | Jeder Unterlauf wird als positiver Impuls von einem Taktzyklus Länge an der entsprechenden Portleitung ausgegeben. |
| OUTMODE = 1 | Toggle-Mode | Bei jedem Unterlauf wechselt die entsprechende Port-Leitung von Low nach High oder von High nach Low. Bei jedem Start des Timers beginnt die Ausgabe mit High. |

Gestartet und gestoppt wird der Timer von dem START-Bit des Kontrollregisters. START = 0 hält den Zähler an, START = 1 startet ihn.

Mit dem RUNMODE-Bit kann man zwischen dem One-shot-Mode und dem Continuous-Mode wählen. Im One-shot Mode

hält der Timer nach jedem Unterlauf an und setzt das START-Bit auf 0 zurück. Im Continuous-Mode beginnt der Zähler nach jedem Unterlauf wieder beim Startwert.

Wie schon erwähnt, wird beim Schreiben in ein Timer-Register der Wert nicht direkt in das Zählregister, sondern in ein Latch geschrieben (auch Vorteiler (Prescaler) genannt, da die Anzahl der Unterläufe pro Sekunde gleich der Zählfrequenz geteilt durch den Wert im Vorteiler ist).

Um eine Übertragung des Werts vom Latch in den Zähler zu erreichen, gibt es folgende Möglichkeiten:

1. Setzen des LOAD-Bits im Kontrollregister. Dies bewirkt ein sogenanntes Force-Load, d.h. unabhängig vom Zählerzustand wird der Wert des Latches in den Zähler übertragen. Das LOAD-Bit ist ein sogenanntes Strobe-Bit. Das bedeutet, daß das Bit nicht gespeichert wird, sondern nur einen einmaligen Vorgang auslöst. Um danach erneut ein Force-Load zu bewirken, muß noch einmal eine 1 in das LOAD-Bit geschrieben werden.
2. Bei jedem Unterlauf des Timers wird das Latch automatisch in den Zähler übertragen.
3. Nach einem Schreibzugriff auf das Timer High-Register bei stehendem Zähler (Stop = 0) wird er ebenfalls automatisch mit dem Wert des Latches geladen. Aus diesem Grund sollte man auch immer die Reihenfolge einhalten: erst Low-Byte, dann High-Byte.

Belegung der Bits des Kontrollregisters A:

Register Nr. 14 / \$E Name: CRA

D7	D6	D5	D4	D3	D2	D1	D0
nicht ver- wen- det	SPMODE 0=Eing. 1=Ausg.	INMODE 0=Takt 1=CNT	LOAD 1=force load (strobe)	RUNMODE 0=cont. 1=one- shot	OUTMODE 0=pulse 1=toggle	PBOn 0=PB6aus 1=PB6an	START 0=aus 1=an

Belegung der Bits des Kontrollregisters B:

Register Nr. 15 / \$F Name: CRB

D7	D6+D5	D4	D3	D2	D1	D0
ALARM 0=TOD 1=Alarm	INMODE 00=Takt 01=CNT 10=Timer A 11=Timer A+ CNT	LOAD 1=force load (strobe)	RUNMODE 0=cont. 1=one- shot	OUTMODE 0=pulse 1=toggle	PBOn 0=PB7aus 1=PB7an	START 0=aus 1=an

Der Zähler (Event-Counter)

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
8 \$8	LSB Event	E7	E6	E5	E4	E3	E2	E1	E0
9 \$9	Event 8-15	E15	E14	E13	E12	E11	E10	E9	E8
10 \$A	MSB Event	E23	E22	E21	E20	E19	E18	E17	E16

Wie schon eingangs erwähnt, unterscheidet sich der 8520 nur durch geringe Änderungen vom 6526. Sämtliche dieser Änderungen betreffen die Funktion der Register 8 - 11. Beim 6526 befand sich hier eine Echtzeituhr (englische Bezeichnung TimeOfDay TOD), die die Tageszeit in Form von Stunden, Minuten und Sekunden in einzelnen Registern bereitstellte. Beim 8520 wurde diese Uhr durch einen einfachen 24-Bit-Zähler ersetzt, den sogenannten Event-Counter. Dadurch tritt eine leichte Begriffsverwirrung auf, da Commodore in den Datenblättern teilweise die alte Bezeichnung TOD auch noch beim 8520 weiterverwendet.

Die Funktion des Event-Counters ist einfach. Er stellt, wie schon erwähnt, einen 24-Bit-Zähler dar. Das bedeutet, sein Wertebereich reicht von 0 bis 16777215 (\$FFFFFFF). Mit jedem positiven Impuls (Wechsel von Low nach High) der TOD-Leitung wird der Zählerstand um eins erhöht. Hat der Zähler schon \$FFFFFFF erreicht, springt er beim nächsten Zählimpuls wieder auf 0 zurück. Der Zähler kann auf einen definierten Stand gesetzt werden, indem man den gewünschten Wert in die Zählerregister schreibt. Das Register 8 enthält die Bits 0-7 des Zählers, das sogenannte LSB (LowestSignificantByte = niederwertigstes Byte), in Register 9 stehen die Bits 8-15 und in Register 10 (\$A) die Bits 16-23, das MSB des Zählerwerts (MostSignificantByte = höchstwertiges Byte).

Bei jedem Schreibzugriff stoppt der Zähler, damit keine Fehler durch einen plötzlichen Übertrag von einem Register ins andere auftreten (s. vorangegangenen Abschnitt). Erst nachdem der Wert ins LSB geschrieben wurde (Reg. 8), läuft der Zähler wieder weiter. Im Normalfall wird man also die Reihenfolge Register 10 (MSB), dann Register 9 und abschließend Register 8 (LSB) einhalten.

Damit beim Lesen des aktuellen Zählerstands keine Übertragfehler auftreten, wird der Zählerwert beim Lesen des MSB (Reg. 10) in ein Latch geschrieben. Jeder weitere Zugriff auf eines der Zählerregister liefert jetzt den Wert des Latches, das somit in aller Ruhe ausgelesen werden kann, während der Zähler intern weiterläuft. Erst beim Lesen des LSBs wird das Latch wieder abgeschaltet. Will man also den Zähler auslesen, sollte man dieselbe Reihenfolge wie beim Schreiben einhalten: erst MSB, dann das Register 9 und am Ende das LSB.

Zusätzlich ist noch eine sogenannte Alarm-Funktion integriert. Setzt man im Kontrollregister B das Alarm-Bit (Bit Nr. 7) auf 1, kann man durch Schreiben in die Register 8-10 einen Alarmwert setzen. Sobald dann der Wert des Zählers mit dem Alarmwert übereinstimmt, wird das Alarm-Bit im Interrupt-Kontrollregister gesetzt. Der Alarmwert kann nur gesetzt werden, ein Lesezugriff auf die Adressen 8-10 ergibt immer den aktuellen Zählerstand,

egal ob das Alarm-Bit im Kontrollregister B gesetzt ist oder nicht.

Der serielle Port

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
12	\$C SDR	S7	S6	S5	S4	S3	S2	S1	S0

Der serielle Port besteht im wesentlichen aus dem seriellen Datenregister (SDR, Serial Data Register) und einem 8-Bit-Schieberegister, auf das man allerdings keinen direkten Zugriff hat. Mit dem SPMODE-Bit im Kontrollregister A kann man zwischen Eingang (SPMODE = 0) und Ausgang (SPMODE = 1) umschalten. Im Eingabemodus werden die seriellen Daten an der SP-Leitung mit jeder positiven Flanke (Wechsel von Low nach High) der CNT-Leitung in das Schieberegister geschoben. Nach acht CNT-Impulsen ist das Schieberegister voll, und sein Inhalt wird in das serielle Datenregister übertragen. Gleichzeitig wird das SP-Bit im Interrupt-Kontrollregister (ICR) gesetzt. Treten jetzt wieder CNT-Impulse auf, werden die Daten weiter in das Schieberegister geschoben, bis dieses erneut voll ist. Hat der Anwender inzwischen das serielle Datenregister (SDR) gelesen, wird der neue Wert ins SDR kopiert, und die Übertragung läuft nach diesem Schema kontinuierlich weiter.

Um den seriellen Port als Ausgang verwenden zu können, setzt man SPMODE auf 1. Die Unterlaufrate von Timer A, der im Continuous-Mode laufen muß, bestimmt die Baudrate (Anzahl der Bits pro Sekunde). Die Daten werden immer mit der halben Unterlaufrate von Timer A aus dem Schieberegister herausgeschoben, wobei die maximale Ausgaberate ein Viertel der Taktfrequenz des 8520 beträgt.

Die Übertragung beginnt, nachdem das erste Datenbyte in das SDR geschrieben wurde. Das CIA überträgt das Datenbyte in das Schieberegister. Die einzelnen Datenbits erscheinen jetzt mit der halben Unterlaufrate von Timer A an der SP-Leitung, das Taktsignal von Timer A liegt dabei an der Leitung CNT an (sie wechselt mit jedem Unterlauf von Timer A ihren Pegel, mit je-

der negative Flanke (Wechsel von High nach Low) erscheint das nächste Bit an der SP-Leitung). Die Übertragung beginnt mit dem MSB des Datenbytes. Sind alle acht Bits ausgegeben, bleibt CNT auf High und die SP-Leitung auf dem Pegel des zuletzt ausgegebenen Bits. Außerdem wird das SP-Bit im Interrupt-Kontrollregister gesetzt, um anzuzeigen, daß das Schieberegister mit neuen Daten versorgt werden kann. Wurde schon vor der Ausgabe des letzten Bits das nächste Datenbyte in das Datenregister geschrieben, wird die Datenausgabe ohne Unterbrechung fortgesetzt.

Will man also eine kontinuierliche Übertragung, muß man das serielle Datenregister rechtzeitig mit neuen Daten versorgen.

Die SP- und die CNT-Leitung sind als Open-Collector-Ausgänge ausgeführt, damit mehrere 8520 über diese Leitungen zusammengeschaltet werden können.

Das Interrupt-Kontrollregister (ICR):

Lesezugriff (read) = Datenregister

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
13	\$D ICR	IR	0	0	FLAG	SP	Alarm	TB	TA

Schreibzugriff (write) = Maskenregister

Register	Name	D7	D6	D5	D4	D3	D2	D1	D0
13	\$D ICR	S/C	x	x	FLAG	SP	Alarm	TB	TA

Das ICR besteht aus einem Datenregister und einem Maskenregister. Jede der fünf Interrupt-Quellen kann ihr korrespondierendes Bit im Datenregisters setzen. Hier noch einmal alle fünf möglichen Interrupt-Quellen:

1. Unterlauf von Timer A (TA, Bit 0)
2. Unterlauf von Timer B (TB, Bit 1)

3. Übereinstimmung des Werts des Event-Counters mit dem Alarmwert (Alarm, Bit 2)
4. Das Schieberegister des Seriellen Ports ist voll (Eingabe) oder leer (Ausgabe) (SP, Bit 3)
5. Negative Flanke am FLAG-Eingang (FLAG, Bit 4)

Liest man das ICR-Register, erhält man immer den Wert des Datenregisters, welches dabei gelöscht wird (alle gesetzten Bits inclusive dem IR-Bit werden zurückgesetzt)! Benötigt man den Wert des Datenregisters noch, muß er nach dem Lesen im RAM gespeichert werden.

Das Maskenregister kann nur beschrieben werden. Sein Wert bestimmt, ob ein gesetztes Bit im Datenregister einen Interrupt auslösen kann. Um einen Interrupt zu ermöglichen, muß das entsprechende Bit des Maskenregisters auf 1 gesetzt werden. Der 8520 legt, sobald ein Bit sowohl im Daten- als auch im Maskenregister gesetzt ist, die IRQ-Leitung auf 0 (da die IRQ-Leitung 0-aktiv ist, wird damit ein Prozessor-Interrupt ausgelöst) und setzt das IR-Bit (Bit 7) im Datenregister, das damit einen Interrupt auch softwaremäßig signalisiert. Erst wenn das ICR gelesen und damit das Datenregister gelöscht wird, springt die IRQ-Leitung wieder auf 1 zurück.

Das Maskenregister kann allerdings nicht wie eine normale Speicherstelle beschrieben werden. Um ein Bit im Maskenregister zu setzen, muß man das gewünschte Bit und das S/C-Bit (Set/Clear, Bit Nr. 7) setzen. Alle übrigen Bits bleiben unbeeinflusst. Um ein Bit zu löschen, muß man ebenfalls das gewünschte Bit setzen, nur bleibt diesmal das S/C-Bit auf 0. Das S/C-Bit bestimmt also, ob die gesetzten Bits der eigenen Maske die entsprechenden Bits des Maskenregisters löschen (S/C = 0) oder setzen (S/C = 1). Alle gelöschten Bits in der eigenen Maske haben keine Auswirkung auf die Bits im Maskenregister. Dazu ein Beispiel:

Es soll ein Interrupt durch die FLAG-Leitung erlaubt werden. Der aktuelle Wert des Maskenregisters soll 00000011 binär betragen, d.h. beide Timerinterrupts sind erlaubt.

Folgender Wert muß ins Maskenregister geschrieben werden: 10010000 binär (S/C = 1, d.h. das gesetzte Bit, in diesem Fall das FLAG-Bit (Bit Nr. 5) setzt das Bit im Maskenregister). Danach hat es folgenden Inhalt:

00010011. Will man jetzt die beiden Timerinterrupts verbieten, schreibt man folgenden Wert: 00000011 (S/C = 0, d.h. die gesetzten Bits (TA und TB, Bit Nr. 0 und 1) werden im Maskenregister gelöscht). Jetzt enthält das Maskenregister 00010000, und nur noch der FLAG-Interrupt ist erlaubt.

Die Einbindung der CIAs in das Amiga-System

Wie eingangs erwähnt, besitzt der Amiga zwei CIAs vom Typ 8520. Die Basisadresse des ersten 8520, kurz 8520-A genannt, ist \$BFE001. Die Register liegen aber nicht durchgehend im Adreßraum, sondern mit Abständen von je 256 Bytes. D.h. alle Register des 8520-A liegen auf ungeraden Adressen, da der 8520-A mit den unteren 8 Leitungen des Prozessordatenbus verbunden ist (D0-7). Folgende Tabelle listet die Adressen der einzelnen Register mit ihrer Verwendung im Amiga auf (näheres über die Bedeutung der einzelnen Portbits im Kapitel Schnittstellen):

Bei Adressen \$BFD000 für CIA-B und \$BFE001 für CIA-A sind die von Commodore angegebenen Basisadressen der CIAs. Bei genauer Untersuchung des Schaltplans stellte sich jedoch heraus, daß die beiden CIAs im gesamten Bereich von A0xxxx bis BFxxxx adressiert werden. Die Wahl zwischen den beiden CIAs treffen die Adreßleitungen A12 und A13. CIA-A wird selektiert, wenn A12 = 0 ist und CIA-B bei A13 = 0, immer vorausgesetzt, die Adressen bewegen sich zwischen A0xxxx und BFxxxx. Da der Datenbus des CIA-A mit den Prozessordatenbus-Leitungen D0-7 (ungerade Adressen) verbunden ist und CIA-B mit D8-15 (gerade Adressen), können beide gleichzeitig in einem Wortzugriff angesprochen werden, wenn A12 und A13 gleich 0 sind.

MOVE.W \$BF0000,D0 lädt die PA-Register beider CIAs in D0, die unteren 8 Bits in D0 enthalten dann den Inhalt des PA-Registers von CIA-A und die Bits 9-15 den des PA-Registers von CIA-B.

Zusammengefaßt kann man folgendes Schema für die Adressierung der CIAs aufstellen: CIA-A wird bei folgender Adresse selektiert (binär):

101x xxxx xxx0 rrrr xxxx xxx1

und CIA-B bei:

101x xxxx xx0x rrrr xxxx xxx0

Die vier mit rrrr bezeichneten Bits wählen das entsprechende Register aus.

Dies gilt allerdings nur für den A1000 mit hundertprozentiger Gewißheit. Es kann durchaus möglich sein, daß sich dies bei den neueren Amiga-Modellen geändert hat. Wer sichergehen will, sollte die von Commodore empfohlenen Adressen benutzen. (CIA-A bei \$BFE001 und CIA-B bei \$BFD000)

Folgende Liste stellt die Belegung der verschiedenen Signalleitungen der CIAs im Amiga dar:

CIA-A:

/IRQ	/INT2-Eingang von Paula
/RES	System-Resetleitung
D0-D7	Prozessordatenbus Bits 0-7
A0-A3	Prozessoradreibbus Bits 8-11
Phi2	E-Takt des Prozessors
R/W	Prozessor R/W
PA7	Gameport 1 Pin 6 (Feuerknopf)
PA6	Gameport 0 Pin 6 (Feuerknopf)
PA5	/RDY "disk ready" Signal vom Diskettenlaufwerk
PA4	/TK0 "disk track 00" Signal vom Diskettenlaufwerk
PA3	/WPRO "write protect" Signal vom Diskettenlaufwerk
PA2	/CHNG "disk change" Signal vom Diskettenlaufwerk
PA1	LED Steuerung der Power-LED (0 = hell, 1 = dunkel)
PA0	OVL memory overlay bit (nicht verändern!)
SP	KDAT serielle Daten von der Tastatur
CNT	KCLK Taktfrequenz für die Tastaturdaten
PB0-PB7	Centronics-Port Datenleitungen
PC	/drdy Centronics-Handshakesignal: Daten bereit
FLAG	/ack Centronics-Handshakesignal: Daten übernommen

CIA-B:

/IRQ	/INT6-Eingang von Paula
/RES	System-Resetleitung
D0-D7	Prozessordatenbus Bits 8-15
A0-A3	Prozessoradressbus Bits 8-11
Phi2	E-Takt des Prozessors
R/W	Prozessor R/W
PA7	/DTR serielle Schnittstelle, /DTR-Signal
PA6	/RTS serielle Schnittstelle, /RTS-Signal
PA5	/CD serielle Schnittstelle, /CD-Signal
PA4	/CTS serielle Schnittstelle, /CTS-Signal
PA3	/DSR serielle Schnittstelle, /DSR-Signal
PA2	SEL "select" Signal zur Centronics-Schnittstelle

PA1	POUT	"paper out" Signal von der Centronics-S.
PA0	BUSY	"busy" Signal von der Centronics-S.
SP	BUSY	direkt mit PA0 verbunden
CNT	POUT	direkt mit PA1 verbunden
PB7	/MTR	"motor" Signal zum Diskettenlaufwerk
PB6	/SEL3	"drive select" für Laufwerk Nr. 3
PB5	/SEL2	"drive select" für Laufwerk Nr. 2
PB4	/SEL1	"drive select" für Laufwerk Nr. 1
PB3	/SEL0	"drive select" für Laufwerk Nr. 0 (intern)
PB2	/SIDE	"side select" Signal zum Diskettenlaufwerk
PB1	DIR	"direction" Signal zum Diskettenlaufwerk
PB0	/STEP	"step" Signal zum Diskettenlaufwerk
FLAG	/INDEX	"index" Signal vom Diskettenlaufwerk
PC		nicht benutzt

1.2.3 Die Customchips und ihre Einbindung in die Amiga-Hardware

Die Chips, von denen wir bis jetzt gehört haben, waren eher banal. Auch der 68000 ist doch trotz seiner unbestrittenen Leistungsfähigkeit heute nur noch ein Standardchip, das für ein paar Mark im Elektronikladen um die Ecke zu haben ist. Obwohl auch die Fähigkeiten des Amiga letztlich von der Geschwindigkeit des Prozessors abhängen, fallen dem staunenden Amiga-Bewunderer doch erst einmal andere Dinge ins Auge. Denn wenn der Computer Grafiken in Fernsehqualität auf den Bildschirm zaubert und dazu noch Beethovens Neunte so aus den Lautsprechern kommt, daß der staunende Freak unwillkürlich den versteckten CD-Player sucht, dann erregt er die Aufmerksamkeit. Ob der Computer dann auch noch sämtliche Primzahlen in Bruchteilen von Sekunden errechnet oder ob er kaum schneller als der alte Taschenrechner ist, interessiert die möglichen Käufer eines Computers wie der Amiga gewöhnlich erst hinterher.

Auch die Entwickler des Amiga richteten sich danach und statteten ihn mit einer für Computer dieser Preisklasse nie gekannten Fülle von grafischen und akustischen Fähigkeiten aus, die der Amiga-Hardware einen geheimnisvollen Flair verliehen. Dies

wurde noch dadurch verstärkt, daß anfangs alle möglichen und unmöglichen Gerüchte über seine Fähigkeiten kursierten.

Das Ziel dieses Kapitels ist es, die Hardware, die für die fantastischen Sound- und Grafikmöglichkeiten des Amiga verantwortlich ist, allgemeinverständlich zu erklären und damit dem Leser eine fundierte Basis für die perfekte Programmierung des Amiga zu geben.

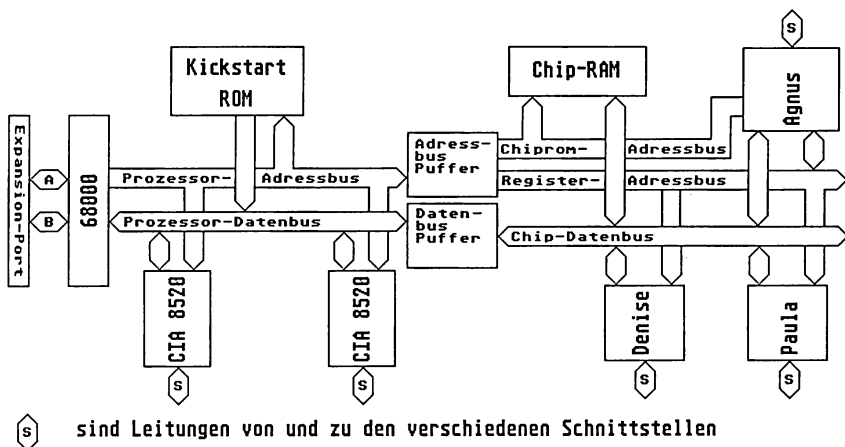
Grundlage aller oben erwähnten Fähigkeiten sind lediglich drei Chips. Sie wurden eigens für den Amiga entwickelt und tragen daher die Bezeichnung Customchips. (Customchips nennt man integrierte Schaltkreise, die von einer Halbleiterfirma eigens für ein bestimmtes Gerät oder einen bestimmten Kunden nach dessen Angaben entwickelt wurden, die deutsche Bezeichnung dafür lautet "Kundenspezifische Schaltkreise".) Ihre Typenbezeichnungen sind 8361, 8362 und 8364, allerdings waren den Amiga-Entwicklern diese Nummern zu farblos, und so gaben sie den drei Chips die Namen Agnus, Denise und Paula. (Im deutschen Amiga steckt allerdings eine an die PAL-Fernsehnorm angepaßte Agnus-Version mit der Typennummer 8367.)

Diese Customchips übernehmen die Aufgaben der Tonerzeugung, der Bildschirmdarstellung, des prozessorunabhängigen Diskettenzugriffs und vieles mehr. Allerdings sind diese Aufgaben nicht so auf die einzelnen Chips verteilt, daß eines die gesamte Tonerzeugung, ein anderes die Grafik und ein weiteres den Diskettenzugriff übernimmt, wie das bei den meisten Konkurrenzprodukten der Fall ist, sondern die Aufgaben sind meistens über mehrere Chips verteilt, so wird z.B. die Grafikdarstellung von zwei Chips zusammen erledigt.

Bei einer solchen engen Zusammenarbeit hätte man die drei Chips auch zu einem einzigen zusammenfassen können, aber die Herstellung eines solchen komplexen Schaltkreises wäre dann teurer gekommen als die Herstellung der drei einzelnen.

Bevor wir jetzt auf die Funktionen von Agnus, Denise und Paula im einzelnen eingehen, erst einmal eine kleine Einführung in den Aufbau des Amiga.

1.2.3.1 Der Grundaufbau des Amiga



Der Grundaufbau des Amiga

Abb. 1.2.3

Ein einfaches Computersystem besteht normalerweise aus einem Prozessor, dem ROM mit dem Betriebssystem, einer gewissen Menge RAM und mindestens einem Peripheriebaustein zur Daten-Ein- und Ausgabe. Sämtliche Bausteine sind mit dem Adreßbus und dem Datenbus verbunden. Der Prozessor kontrolliert das System, nur er kann Adressen auf den Bus legen und damit Daten an bestimmte Systemkomponenten, z.B. das RAM, ausgeben oder von ihnen einlesen. Er kontrolliert auch die Bussteuersignale wie z.B. die R/W-Leitung. (Sie sind der Einfachheit halber in Abbildung 1.2.3 nicht eingezeichnet; die einzelnen Bussteuersignale des 68000 sind in Kapitel 1.2.1 aufgeführt.)

Auch enthält jedes Computersystem noch Steuerschaltungen wie z.B. einen Adreßdecoder, der bei bestimmten Werten auf dem Adreßbus die diesen Adressen zugewiesenen Bausteine aktiviert.

Doch zurück zum Amiga. Wie man aus der Abbildung 1.2.3 ersehen kann, weicht der Aufbau des Amiga doch etwas vom oben

beschriebenen ab. Auf der linken Seite sehen wir den 68000-Mikroprozessor, dessen Daten- und Adreßleitungen direkt mit den beiden CIAs vom Typ 8520 und dem Kickstart-ROM verbunden sind. Dieser Teil des Amiga-Systems ist also konventionell aufgebaut, nur der Prozessor hat Zugriff auf die beiden CIAs und das ROM. Wie sieht es nun auf der rechten Seite aus? Hier befinden sich die drei Customchips Agnus, Denise und Paula und das Chip-RAM, die alle direkt mit demselben Datenbus verbunden sind, welcher aber vom Prozessordatenbus durch einen Puffer getrennt ist, der wahlweise den Prozessordatenbus mit dem Chipdatenbus verbinden oder von ihm trennen kann. Die drei Customchips sind untereinander auch noch durch den Registeradreibus verbunden, der wie der Datenbus wahlweise mit dem Prozessoradreibus verbunden werden kann oder nicht.

Da das Chip-RAM einen weitaus größeren Adreßbereich hat als die Customchips und außerdem gemultiplexte Adressen benötigt, existiert noch ein Chip-RAM-Adreibus. Für diejenigen, denen der Begriff "gemultiplexte Adressen" nichts sagt, nur soviel: die RAM-Chips, die im Amiga (A1000) verwendet werden, haben einen Adreßbereich von 2^{16} Adressen (65536). Um alle Adressen eines Chips ansprechen zu können, benötigt man also 16 Adreßleitungen. Da die eigentlichen Chips aber sehr klein sind, hätte eine solch große Anzahl von Adreßleitungen ein unverhältnismäßig großes Gehäuse zur Folge. Um dieses Problem zu beseitigen, hat man eine sogenannte gemultiplexte Adressierung eingeführt. Das Gehäuse hat dabei lediglich acht Adreßleitungen, auf denen hintereinander erst die oberen acht Adreßbits und danach die unteren acht übertragen werden. Das Chip speichert die oberen acht und hat dann, wenn die unteren an den Adreßleitungen anliegen, alle 16 Adreßbits, die es benötigt.

Wieder zum Amiga: Warum diese Trennung der beiden Busse? Nun, der Grund dafür ist, daß die verschiedenen Ein-/Ausgabegeräte eine kontinuierliche Versorgung mit Daten benötigen. So müssen z.B. die Daten für die einzelnen Bildpunkte auf dem Monitor fünfzigmal in der Sekunde aus dem RAM gelesen werden, da ein Fernsehbild nach der deutschen Pal-Fernsehnorm fünfzigmal in der Sekunde neu aufgebaut wird.

Eine hochauflösende Grafik kann auf dem Amiga über 64 KB Bildschirmspeicher benötigen. Dies bedeutet, daß pro Sekunde $50 * 64$ KB aus dem Speicher geholt werden müssen. Dies sind etwas über 1.5 Millionen Speicherzugriffe in jeder Sekunde! Müßte der Prozessor diese Aufgabe erledigen, wäre er hoffnungslos überfordert. Eine so hohe Datenrate schafft auch ein 68000 nicht. Und dabei ist der Amiga auch noch in der Lage, neben der Grafik digitalisierte Tonausgabe und Diskettenzugriffe durchzuführen, ohne den 68000 zu verwenden. Die Lösung liegt in einem zweiten Prozessor, der all diese Speicherzugriffe selbständig ausführt. Einen solchen Prozessor nennt man auch DMA-Controller (Direct Memory Access / Direkter Speicherzugriff). Er ist beim Amiga in Agnus untergebracht. Aus diesem Grund ist Agnus auch mit dem Chip-RAM-Adreßbus verbunden.

Die übrigen beiden Chips, Denise und Paula, und auch der Rest von Agnus sind wie herkömmliche Peripheriechips aufgebaut. Sie besitzen eine gewisse Anzahl von Registern, die vom Prozessor (oder dem DMA-Controller) gelesen oder beschrieben werden können. Die einzelnen Register werden über den Registeradreßbus ausgewählt. Er hat acht Leitungen, kann also 256 verschiedene Zustände annehmen. Es gibt keine spezielle Auswahl der einzelnen Chips. Hat der Adreßbus den Wert 255 oder \$FF, sind also alle Leitungen High, ist kein Register ausgewählt. Liegt aber eine gültige Registernummer an, erkennt dies das Chip, welches das gewählte Register enthält, und aktiviert dieses. Diese Aufgabe wird in den einzelnen Chips von dem Register-Adreßdecoder ausgeführt. Dadurch, daß die Auswahl eines Registers nur von seiner Registeradresse abhängt und nicht von dem Chip, in dem es sich befindet, können auch zwei Register in zwei verschiedenen Chips gleichzeitig mit demselben Wert beschrieben werden, wenn diese auch dieselbe Registeradresse haben. Von dieser Möglichkeit wird bei einigen Registern Gebrauch gemacht, die Daten enthalten, die von mehreren Chips benötigt werden.

Jedes Chip-Register kann entweder ein Leseregister oder ein Schreibregister sein. Eine Umschaltung zwischen Lesen und Schreiben mittels einer speziellen R/W-Leitung, wie z.B. beim

8520, gibt es nicht. Die Registeradresse bestimmt gleichzeitig, ob ein Lese- oder Schreibzugriff stattfindet. Register, die sich sowohl lesen als auch beschreiben lassen, sind so realisiert, daß der Schreibzugriff auf einer und der Lesezugriff auf einer anderen Registeradresse stattfindet. Dieser Sachverhalt spiegelt sich deutlich in der Liste der Chip-Register wieder (siehe Kapitel 1.5.1).

Da Agnus den DMA-Controller enthält, kann es auch selber auf Register der Customchips zugreifen, d.h. eine Adresse auf dem Registeradreibus ausgeben.

Ein offensichtliches Problem ist aber noch ungelöst. Es gibt nur einen Daten- und einen Adreibus, auf den sowohl der Prozessor als auch der DMA-Controller zugreifen wollen. Einen Bus kann aber immer nur ein Buscontroller steuern. Würden zwei Chips gleichzeitig versuchen, eine Adresse auf den Bus zu legen, würde es zu einer Datenkollision, gefolgt von einem Systemzusammenbruch, kommen. Also müssen sie sich die Buszugriffe teilen und dürfen nur noch abwechselnd auf den Bus zugreifen, wobei natürlich jeder möglichst oft den Bus für sich haben möchte. Dieses Problem wurde beim Amiga auf elegante Art in drei Stufen gelöst:

Als erstes hat man im Amiga die beiden normalerweise durchgehenden Busse in zwei Teile geteilt. Der eine (in der Abbildung links) verbindet all die Bausteine, die nur vom Prozessor angesprochen werden. Bei einem Zugriff des 68000 auf einen dieser Bausteine unterbrechen die beiden Puffer (in der Mitte der Abbildung) die Verbindungen des Prozessordaten- und des Prozessoradreibus mit dem Chipdaten- und Chipadreibus. Dadurch können sowohl der Prozessor auf seiner Seite und Agnus auf der anderen ungestört auf den Bus zugreifen. Der Prozessor hat dadurch einen ungestörten Zugriff auf das Betriebssystem und eventuelle RAM-Erweiterungen, die am Expansion-Port angeschlossen werden. Dieses Erweiterungs-RAM wird deshalb auch Fast RAM genannt, da der Prozessor immer ohne Geschwindigkeitsverlust darauf zugreifen kann. (Die RAM-Erweiterungen, die beim A1000 vorne und beim A500 an der Gehäuseunterseite eingesteckt werden, gehören allerdings noch zum Chip-RAM.)

Als zweites hat man die Buszugriffe vom Prozessor und von Agnus ineinander verschachtelt, so daß normalerweise auch bei Zugriffen auf das Chip-RAM oder die Chip-Register der 68000 nicht gebremst werden muß. Bei einem solchen Zugriff verbinden die Puffer die beiden Bussysteme wieder.

Erst als dritter und letzter Ausweg werden die Zugriffe des Prozessors verzögert, d.h. der Prozessor muß warten, bis Agnus seine DMA-Zugriffe beendet hat und der Bus wieder frei ist. Dieser Fall tritt allerdings nur dann ein, wenn entweder sehr hohe Grafikauflösungen gewählt wurden oder der Blitter in Betrieb war. Näheres dazu aber später. Jetzt soll erst einmal der interne Aufbau der drei Customchips erläutert werden.

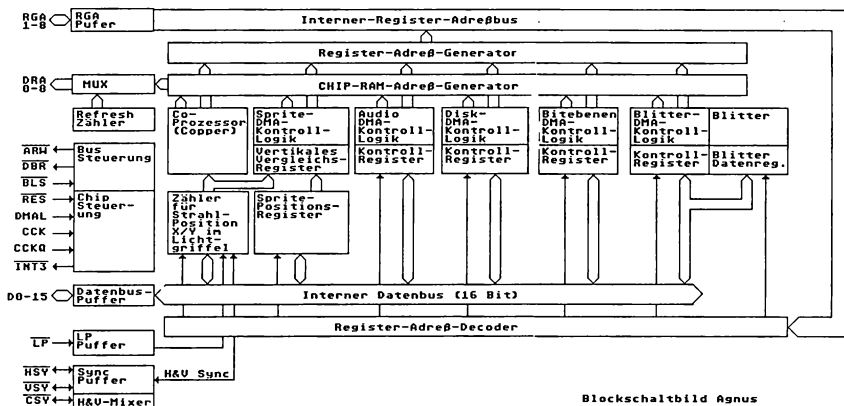


Abb. 1.2.3.2

Wie schon oben erwähnt, enthält Agnus die gesamte DMA-Steuerung. Für jede der 6 möglichen DMA-Quellen existiert eine eigene Kontrolllogik. Sie sind alle sowohl mit dem Chip-RAM-Adreßgenerator als auch dem Registeradreßgenerator verbunden. Diese Adreßgeneratoren erzeugen die RAM-Adresse der gewünschten Chip-RAM-Speicherstelle und die Registeradresse des Zielregisters. Auf diese Weise versorgen die DMA-Logiken die entsprechenden Chip-Register mit Daten aus dem RAM oder schreiben den Inhalt bestimmter Register ins RAM.

Verbunden mit dem Chip-RAM-Adreßgenerator ist auch noch ein Refreshzähler, der die für den Betrieb der dynamischen RAM-Chips notwendigen Refresh-Signale erzeugt.

Agnus steuert auch den zeitlichen Ablauf der einzelnen DMA-Zugriffe. Als Grundlage dazu dient eine Bildschirmzeile. In jeder Bildschirmzeile finden 225 Speicherzugriffe statt, die von Agnus auf die einzelnen DMA-Kanäle und den 68000 verteilt werden. Da Agnus dafür immer die aktuelle Zeilen- und Spaltenposition benötigt, enthält es auch den Rasterzeilen- und Spaltenzähler. Diese Zähler für die Strahlposition erzeugen auch die horizontalen und vertikalen Synchronisationssignale, die dem angeschlossenen Monitor den Anfang einer neuen Zeile (H-Sync)

und den eines neuen Bildes (V-Sync) signalisieren. Die horizontalen und vertikalen Synchronsignale können auch von außen in Agnus eingespeist werden und steuern dann die internen Rasterzeilen- und Spaltenzähler. Dadurch kann das Videobild des Amiga mit einem anderen, z.B. von einem Videorecorder, synchronisiert werden. Dieses sogenannte Genlock kann dadurch beim Amiga einfach realisiert werden. (Das Synchronisieren zweier Videobilder bedeutet, vereinfacht gesagt, daß die einzelnen Rasterzeilen und die einzelnen Bilder beider Signale gemeinsam anfangen.)

Zwei weitere wichtige Elemente von Agnus sind der Blitter und der Coprozessor Copper. Der Blitter ist eine spezielle Schaltung, die in der Lage ist, Speicherbereiche zu manipulieren oder zu verschieben. Er kann dadurch den 68000 entlasten, da er diese Aufgaben weitaus schneller ausführen kann. Außerdem ist der Blitter noch in der Lage, selbständig Linien zu zeichnen und Flächen zu füllen. Der Copper ist ein einfacher Coprozessor. Seine Programme, die sogenannten Copperlisten, enthalten nur drei verschiedene Befehle. Der Copper ist in der Lage, die verschiedenen Chip-Register zu festgelegten Zeitpunkten beliebig zu verändern. Genauere Informationen über den Blitter und den Copper finden sich in den Kapiteln über die Programmierung der Customchips.

Hier noch eine Funktionsbeschreibung der einzelnen Pins:

Datenbus: D0-D15

Die 16 Datenleitungen sind direkt mit dem Chip-RAM-Datenbus verbunden. Intern hängen nach einem Puffer sämtliche Chip-Register an dem Bus.

Registeradreibus: RGA0-RGA8 (ReGisterAdreß)

Der Registeradreibus von Agnus ist bidirektional. Bei einem DMA-Zugriff legt der Registeradreibgenerator die gewünschte Registeradresse auf diese Busleitungen. Greift der Prozessor dagegen auf die Chip-Register zu, wirken diese Leitungen als Eingänge, und die vom Prozessor gewählte Registeradresse gelangt

zum Registeradreßdekodeer innerhalb von Agnus. Allgemein gilt, daß bei einem Wert von \$FF auf dem Registeradreßbus, d.h. alle Leitungen sind auf High, kein Register ausgewählt wurde.

Die Adreßleitungen für das dynamische RAM: DRA0-DRA8 (Dynamic RAM Adress / Dynamische RAM-Adresse)

Diese Adreßleitungen sind mit dem Chip-RAM-Adreßbus verbunden. Sie sind reine Ausgänge und werden von Agnus immer dann aktiviert, wenn es einen DMA-Zugriff auf das Chip-RAM ausführen will. Die Adressen auf diesen Pins sind schon gemultiplext und können direkt mit den Adreßleitungen dynamischer 32 Kbit-RAMs verbunden werden (Typ 41256). Dies ist beim Amiga 500 und 2000 der Fall. Beim älteren A1000 besitzen die RAMs nur 8 Adreßleitungen. Die neunte DRA-Leitung von Agnus wird wieder demultiplext und zur Umschaltung zwischen verschiedenen RAM-Banks verwendet.

Die Taktleitungen des Agnus CCK und CCKQ: (Color Clock und Color Clock Delay)

Diese beiden Leitungen sind die einzigen Taktleitungen des Amiga. Die Frequenz beider Signale beträgt 3.58 MHz, das ist die halbe Prozessor-Taktfrequenz. Das Signal CCKQ ist zu dem Signal CCK um einen viertel Taktzyklus (90 Grad) verzögert. Nach diesen beiden Signalen richtet sich die gesamte Zeitsteuerung (das Timing) des Agnus.

Die Bussteuerungssignale von Agnus: BLS, ARW, DBR

Diese drei Leitungen sind mit der Steuerlogik des Amiga verbunden. Mit der Leitung DBR (Data Bus Request / Datenbusanforderung) teilt Agnus dieser Steuerlogik mit, daß er den Bus im nächsten Buszyklus übernehmen wird. Diese Leitung hat immer Vorrang vor einer Busanfrage des Prozessors. Benötigt Agnus den Bus in mehreren aufeinanderfolgenden Buszyklen, muß der 68000 eben warten.

Die Leitung ARW (Agnus RAM Write) signalisiert der Steuerlogik, daß Agnus einen Schreibzugriff auf das Chip-RAM ausführen will.

Das BLS-Signal (Blitter Slow Down / Blitter verlangsamen) signalisiert Agnus, das der Prozessor schon seit drei Buszyklen auf einen Zugriff wartet. Je nach dem internen Zustand überläßt Agnus dem Prozessor dann den Bus für einen Zyklus. Genauerer über die verschiedenen Buszyklen findet sich im Kapitel über die Programmierung der Customchips.

Die Steuersignale: RES, INT3, DMAL

Das RES-Signal (Reset) ist direkt mit der Reset-Leitung des Prozessors verbunden und setzt Agnus in einen definierten Grundzustand zurück.

Die INT3-Leitung (Interrupt der Ebene 3) ist ein Ausgang und direkt mit der gleichnamigen Leitung von Paula verbunden. Agnus signalisiert damit der Interruptlogik von Paula, daß eine Komponente von Agnus einen Interrupt ausgelöst hat.

Die DMAL-Leitung (DMA Request Line / DMA-Anforderung) verbindet ebenfalls Agnus mit Paula. Nur diesmal in umgekehrter Richtung. Paula signalisiert damit, daß Agnus einen DMA-Transfer einleiten soll.

Die Leitungen: HSY, VSY, CSY und LP

Im Normalfall erscheinen an den Leitungen HSY (Horizontal Sync / Horizontales Synchronisationssignal) und VSY (Vertical Sync / Vertikales Synchronisationssignal) die Synchronsignale für den angeschlossenen Monitor. Das Signal an der Leitung CSY (Composite Sync / Kombiniertes Synchronsignal) ist ein Summensignal aus HSY und VSY und dient sowohl dem Anschluß von Monitoren, die ein gemischtes Synchronsignal benötigen, als auch der Schaltung, die das Videosignal erzeugt, dem Videomixer.

Die LP-Leitung (Light Pen) ist ein Eingang und erlaubt den Anschluß eines Lichtgriffels oder Lightpens. Bei einer negativen Flanke an diesem Pin wird der Inhalt des Rasterzählerregisters gespeichert. (siehe Kapitel 1.5.2)

Die HSY- und VSY-Leitungen können auch als Eingang arbeiten und erlauben dann eine externe Synchronisation von Agnus (Genlock).

1.2.3.2 Der Aufbau von Denise

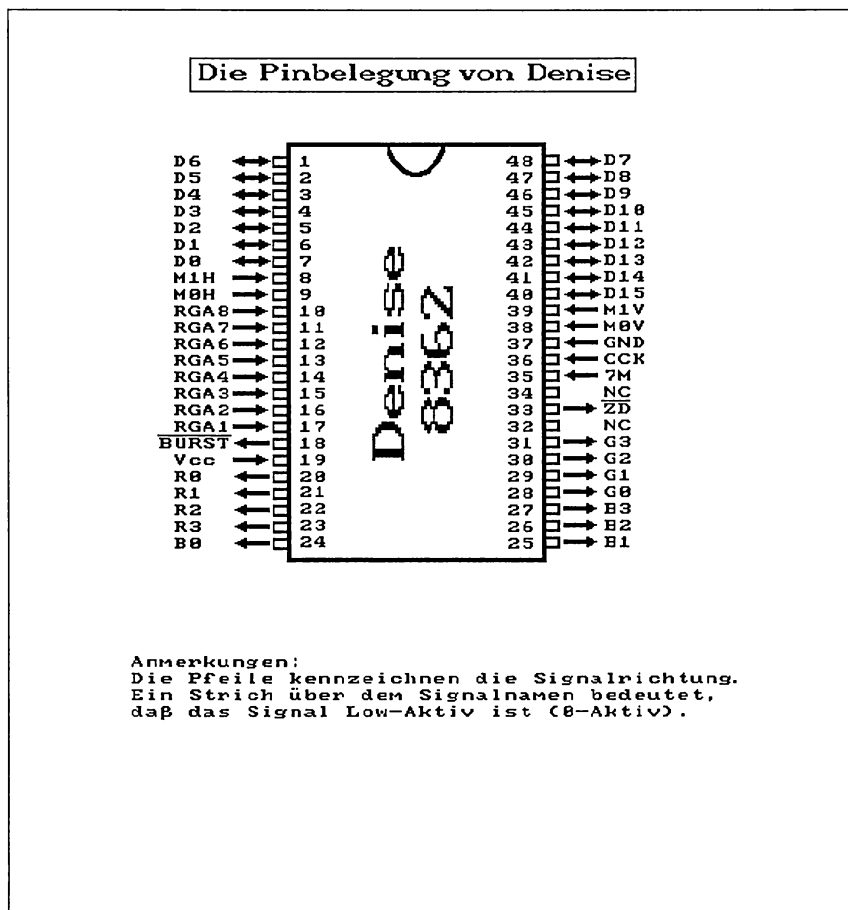


Abb. 1.2.3.3

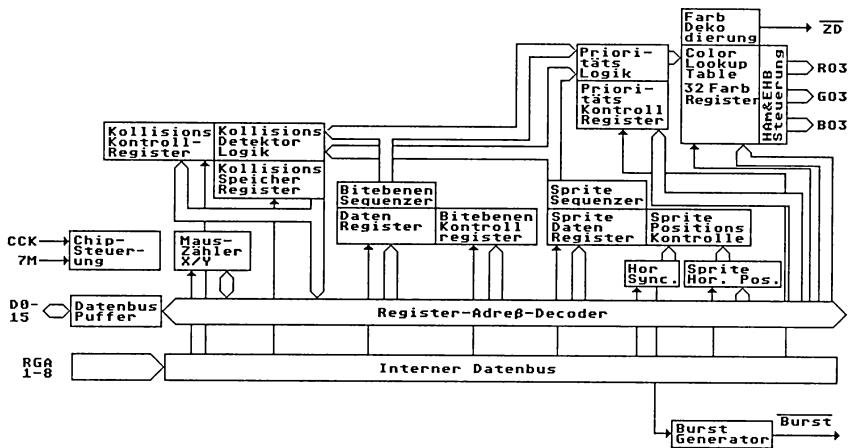


Abb. 1.2.3.4

Die Funktion von Denise kann man ganz allgemein mit Bild-erzeugung umschreiben. Der erste Teil dieser Arbeit wird allerdings schon von Agnus erledigt. Dieser holt die aktuellen Grafikdaten aus dem Chip-RAM und schreibt sie in die für die Bitebenen zuständigen Register von Denise. Ebenso verfährt Agnus mit den Spritedaten. Denise enthält also immer sämtliche Grafik und Spritedaten für 16 Punkte, da immer ein Bit einem Punkt auf dem Bildschirm entspricht und die Datenregister alle eine Breite von einem Wort, also 16 Bit, haben. Diese Daten müssen von Denise in die entsprechende RGB-Darstellung umgewandelt werden. Als erstes werden die Grafikdaten von einer parallelen 16-Bit-Darstellung mittels des Bitebenen-Sequenzer in einen seriellen Datenstrom umgewandelt. Da maximal 6 Bitebenen möglich sind, gibt es diesen Funktionsblock ebenfalls 6mal. Die seriellen Datenströme der einzelnen Bitebenen-Sequenzer werden jetzt zu einem maximal 6 Bit breiten Datenstrom zusammengefaßt.

Die Prioritäts-Kontrollogik wählt aus den Grafikdaten von den Bitebenen-Sequenzern und den Spritedaten aus den Sprite-Sequenzern anhand der eingestellten Prioritäten die für den aktuellen Punkt gültigen Daten aus. Nach diesen Daten selektiert

die Farbdecodierung eines der 32 Farbregister. Der Wert dieses Farbregisters wird dann als digitales RGB-Signal ausgegeben. Wenn der Hold-And-Modify (HAM)- oder der Extra-Half-Bright-(EHB)-Modus gewählt wurde, werden die Daten aus dem Farbregister noch dementsprechend modifiziert, bevor sie das Chip verlassen.

Die Daten von den Sequenzern gelangen auch noch zu der Kollisionskontrolllogik, die, wie ihr Name schon sagt, die Daten auf eine Kollision zwischen den Bitebenen und den Sprites überprüft und das Ergebnis dieser Prüfung in dem Kollisionsregister ablegt.

Die letzte Funktion von Denise hat nichts mit der Bildschirmdarstellung zu tun. Denise enthält auch noch die Maus-Zähler, die die aktuellen X- und Y-Positionen der angeschlossenen Mäuse enthalten.

Hier die Funktionsbeschreibung aller Pins von Denise:

Der Datenbus: D0-D15

Die 16 Datenbusleitungen sind wie die von Agnus mit dem Chipdatenbus verbunden.

Registeradreibus: RGA1-RGA8

Der Registeradreibus ist bei Denise ein reiner Eingang. Mit Hilfe des Werts am Registeradreibus wählt der Register-Adreibusdecoder das entsprechende interne Register aus.

Die Takteingänge: CCK und 7M

Die Timing von Denise richtet sich nach dem CCK-Signal. Der CCK-Pin ist mit der gleichnamigen Leitung von Agnus verbunden. Das Taktsignal auf der 7M-Leitung (7 Megahertz) hat eine Frequenz von 7.15909 MHz. Der Denise-Chip benötigt diese zusätzliche Frequenz für die Ausgabe der einzelnen Bildpunkte, da die Punktfrequenz über den 3.58 MHz des CCK-Signals liegt. Ein Punkt im niedrig auflösenden Modus (320 Punkte/Zeile) hat

genau die Dauer eines 7M-Taktzyklus. Im hochauflösenden Modus (640 Punkte/Zeile) werden in jedem 7M-Takt zwei Punkte ausgegeben. Einer mit jeder Flanke von 7M. Der 7M-Takt ist übrigens auch der Takt des 68000. Er ist mit dessen Clock-Eingang verbunden.

Die Ausgangssignale: R0-3, G0-3, B0-3, ZD und BURST

Die Leitungen R0-3, G0-3 und B0-3 stellen das RGB-Ausgangssignal von Denise dar. Denise gibt den entsprechenden RGB-Wert digital aus. Dabei wird jede der drei Farbkomponenten mit Hilfe von vier Bits dargestellt. Das macht 16 Werte pro Komponente und $16 \cdot 16 \cdot 16$ Farben (4096) insgesamt. Die drei Farbsignale laufen, nachdem sie Denise verlassen haben, erst über einen Puffer und werden dann mittels dreier Digital-/Analog-Wandler in ein RGB-Analogsignal verwandelt, das dann am RGB-Port zur Verfügung steht.

Ein zusätzlicher Videomischer macht aus diesem RGB-Analogsignal das Videosignal für den Videoanschluß. Er benötigt dazu auch noch das BURST-Signal von Denise. Das BURST-Signal ist eine Schwingung mit der Frequenz von CCK, also 3.58 MHz. Näheres über die Funktion des Colorbursts entnehmen Sie am besten einem Buch über Fernsehtechnik.

Das letzte Ausgangssignal von Denise ist das ZD-Signal (Zero Detect oder auch Background Indicator / Hintergrund-Erkennung). Es ist immer dann Low, wenn ein Punkt in der Hintergrundfarbe dargestellt wird, d.h. seine Farbe aus dem Farbreister Nummer 0 stammt. Dieses Signal wird in einem sogenannten Genlock-Adapter benutzt und dient in diesem zur Umschaltung zwischen dem externen Video-Signal, wenn $ZD = 0$, und dem Videosignal des Amiga, wenn $ZD = 1$ ist. Das ZD-Signal liegt ebenfalls am RGB-Port an. Näheres dazu im Kapitel Schnittstellen.

Die Maus-/Joystick-Eingänge: M0H, M1H, M0V, M1V

Diese vier Leitungen entsprechen direkt den Mauseingängen der beiden Gameports (oder Joystickbuchsen). Da der Amiga aber zwei Gameports hat, müßten es eigentlich acht Eingänge sein. Anscheinend waren aber nur noch vier Pins von Denise frei, als man diesen entwickelt hat, und so wurde bei Commodore folgender Weg gewählt: Die acht Eingangsleitungen der beiden Gameports gelangen auf einen Umschalter, der wahlweise die vier Leitungen des vorderen oder des hinteren Ports auf die vier Eingänge von Denise schaltet. Diese Umschaltung geschieht synchron zum Takt von Denise, so daß dieser die vier Leitungen intern wieder auf zwei Register aufteilen kann. Eines für jeden Gameport. Näheres über die Gameports siehe Kapitel Schnittstellen.

1.2.3.3 Der Aufbau von Paula

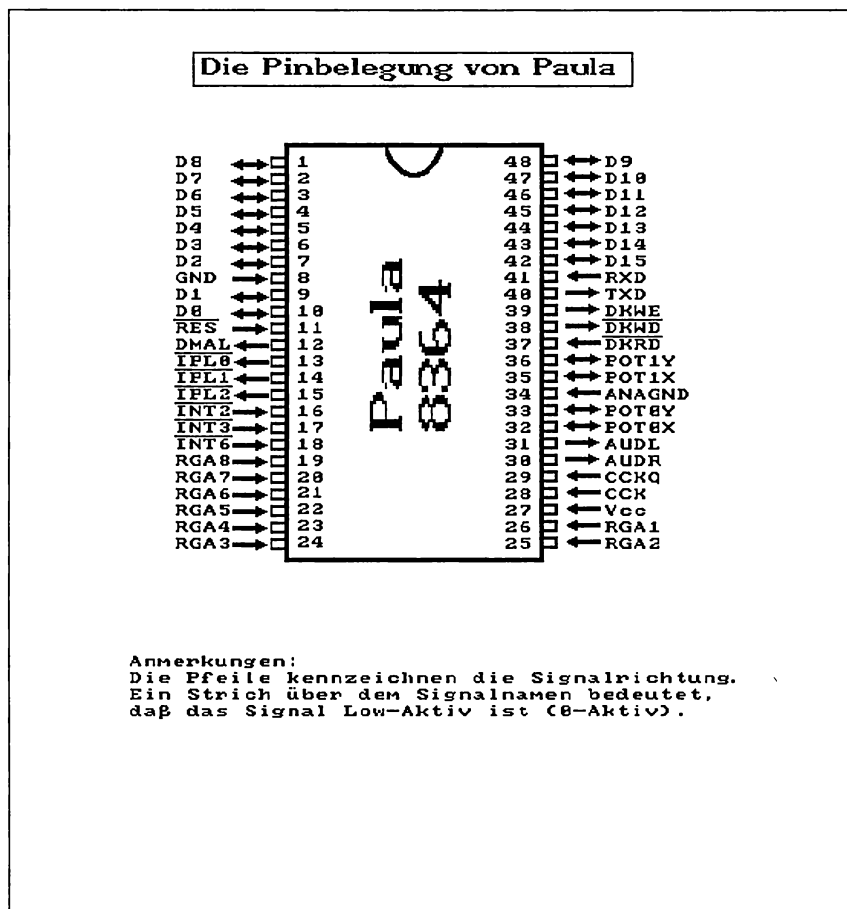


Abb. 1.2.3.5

Blockschaltbild Paula

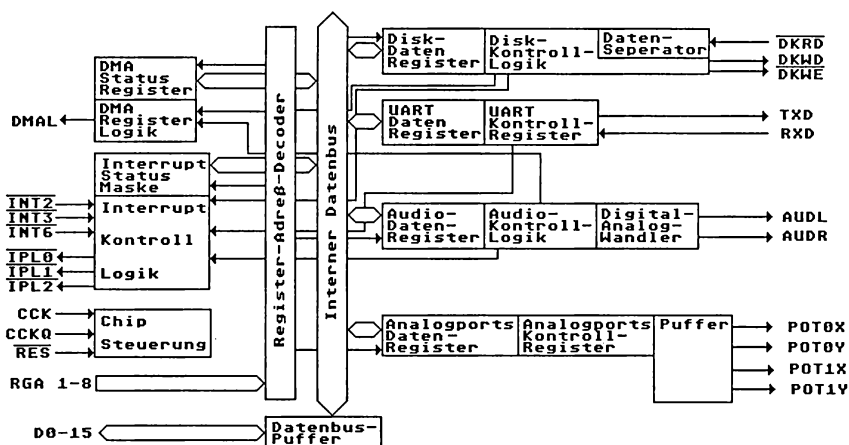


Abb. 1.2.3.6

Die Aufgaben Paulas fallen hauptsächlich in den Ein-/Ausgabe-Bereich (Input/Output kurz I/O), nämlich die Disketten-Ein-/Ausgabe, die serielle Ein-/Ausgabe, die Tonausgabe und die Abfrage der Analogeingänge. Zusätzlich unterliegt Paula die gesamte Interrupt-Steuerung. In ihm laufen alle im System aufgetretenen Interrupts ein. Aus diesen vierzehn möglichen Interrupt-Quellen erzeugt Paula die Interrupt-Signale für den 68000. Es werden dabei Interrupts der Ebenen 1-6 auf den IPL-Leitungen des 68000 generiert. Paula gibt dem Programmierer die Möglichkeit, jede der vierzehn Interrupt-Quellen einzeln zu erlauben oder zu verbieten.

Der Diskdatentransfer und die Tonausgabe laufen ebenfalls per DMA ab. Bei den Daten von Diskette läßt sich nicht immer voraussagen, wann das nächste Datenwort für einen DMA-Transfer von Agnus bereit ist. Ursachen hierfür sind unter anderem die unvermeidlichen Drehzahlschwankungen des Diskettenlaufwerks. Auch bei der Audioausgabe weiß Agnus nicht im voraus, wann Daten benötigt werden. Um dennoch einen reibungslosen DMA-Transfer zu ermöglichen, besitzt Paula die

DMAL-Leitung, mit der es Agnus mitteilen kann, daß es einen DMA-Zugriff benötigt.

Die serielle Kommunikation wird von einem UART-Baustein innerhalb Paulas übernommen. UART heißt Universal Asynchronous Receive Transmit was soviel wie universeller asynchroner Sender/Empfänger bedeutet.

Die Funktion des UARTs wird genauso wie die der vier Audiokanäle und der Analogports später im Kapitel über die Programmierung der Customchips beschrieben.

Wie üblich zum Abschluß noch eine Beschreibung der Pinfunktionen:

Datenbus: D0-15

Wie bei den anderen Chips mit dem Chipdatenbus verbunden.

Registeradreibbus: RGA 1-8

Wie bei Denise

Die Taktsignale und Reset: CCK, CCKQ und RES

Paula enthält dieselben Taktsignale wie Agnus. Die Reset-Leitung RES versetzt das Chip in einen definierte Einschaltzustand.

DMA-Request: DMAL

Mittels dieser Leitung signalisiert Paula an Agnus, daß es einen DMA-Transfer benötigt.

Audioausgänge: AUDL und AUDR

Die Ausgängen AUDL und AUDR (Left Audio und Right Audio) sind analoge Ausgänge, an denen die in Paula erzeugten Tonsignale anliegen. An AUDL liegen die internen Tonkanäle 0 und 3, an AUDR die Kanäle 1 und 2.

Die Leitungen der seriellen Schnittstelle: TXD und RXD

RXD (Receive Data / Empfangsdaten) ist der serielle Eingang des UARTs, TXD (Transmit Data / Sendedaten) der serielle Ausgang. Diese Leitungen haben TTL-Pegel, das heißt, ihre Ein-/Ausgangsspannungen bewegen sich zwischen 0 und 5 Volt. Ein zusätzlicher Pegelwandler erzeugt erst nachträglich die +12/-5 Volt der seriellen RS232-Schnittstelle des Amiga.

Die Analogeingänge: POT0X, POT0Y, POT1X, POT1Y

Die Eingänge POT0X und POT0Y sind mit den entsprechenden Leitungen von Gameport 0 verbunden, POT1X und POT1Y dementsprechend mit Port 1. An diese Eingänge können Paddles oder Analog-Joysticks angeschlossen werden. Diese Eingabegeräte enthalten veränderliche Widerstände, sogenannte Potentiometer, die zwischen +5 Volt und den POT-Eingängen liegen. Paula ist in der Lage, den Wert dieser Widerstände zu ermitteln und in internen Registern abzulegen. Die POT-Eingänge können allerdings softwaremäßig auch als Ausgang geschaltet werden.

Die Diskettenleitungen: DKRD, DRWD, DKWE

Über die DKRD-Leitung (Disk Read / Disketten-Lesedaten) erhält PAULA die Lesedaten von der Diskette. Die DKWD-Leitung (Disk Write / Disketten-Schreibdaten) ist der Ausgang für die Daten zum Diskettenlaufwerk. Die DKWE-Leitung (Disk Write Enable / Disketten-Schreibfreigabe) dient zum Umschalten des Diskettenlaufwerks von Lesen auf Schreiben.

Die Interruptleitungen: INT2, INT3, INT6 und IPL0, IPL1, IPL2

Über die drei INT-Leitungen erhält Paula die Aufforderung, einen Interrupt der entsprechenden Ebene zu erzeugen. Mit der INT2-Leitung ist normalerweise der mit CIA-A bezeichnete 8520-Baustein verbunden. Allerdings ist diese Leitung auch noch zum Expansion-Port und der seriellen Schnittstelle durchgeschleift. Wird sie Low, erzeugt Paula einen Interrupt der Ebene 2, vorausgesetzt, daß ein Interrupt dieser Ebene überhaupt erlaubt ist. Die INT3-Leitung ist mit dem entsprechenden Ausgang von Agnus verbunden und die INT6-Leitung mit dem CIA-B

und ebenfalls dem Expansion-Port. Alle restlichen Interrupts treten innerhalb der I/O-Komponenten von Paula auf.

Die IPL0-IPL2 Leitungen (Interrupt Pending Level des 68000, s. Kap. 1.2.1) sind direkt mit den entsprechenden Prozessorleitungen verbunden. Über sie erzeugt Paula einen Prozessor-Interrupt der entsprechenden Ebene.

1.2.3.4 Besonderheiten des A500

Die Beschreibungen der Amiga-Hardware in diesem Kapitel entstanden ursprünglich für den A1000. Sie sind größtenteils auch für den A500 gültig. Im A500 wurde am Amiga-Grundaufbau nichts geändert. Man hat nur versucht, eine billigere Version herzustellen. Die größten Unterschiede zwischen beiden Modellen betreffen die Verteilung der verschiedenen Hardware-Elemente auf die einzelnen Chips.

Beim A1000 kommen zu den Customchips noch eine Vielzahl einfacher logischer Schaltkreise, die der Erzeugung der Takt-signale, der Bussteuerung und der Adreßdecodierung dienen.

Beim A500 wurden fast alle dieser logischen Funktionen in den großen Chips zusammengefaßt. Dazu hat man einige neue Bau-gruppen zum Agnus-Chip hinzugefügt und dem neuen Chip den Namen Fat-Agnus gegeben (Typenbezeichnung: 8370).

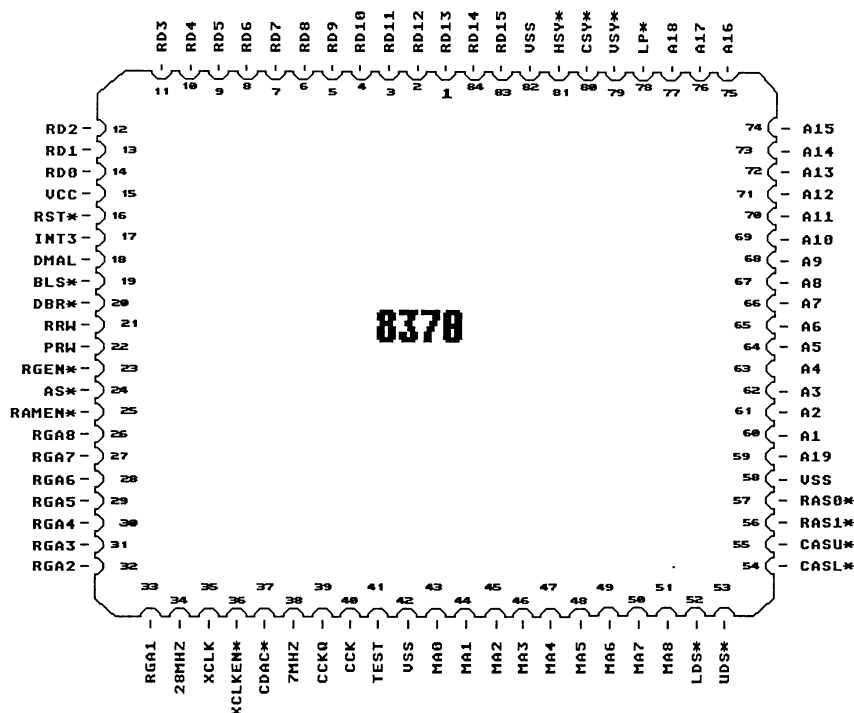


Abb. 1.2.3.7

Die Abbildung 1.2.3.7 zeigt die Pinbelegung des Fat-Agnus. Wenn unter der Vielzahl von Pins einige des normalen Agnus nicht mehr auftauchen, liegt das daran, dass diese mit Schaltungen verbunden waren, die sich jetzt im Inneren von Fat-Agnus befinden. Im einzelnen wurden folgende neue Funktionen in Agnus integriert:

Die Takterzeugung

Die gesamte Takterzeugung für das Amiga-System ist in Fat-Agnus integriert. Lediglich der 28 MHz Haupttakt muß noch zugeführt werden. Die zu diesem Funktionsblock gehörenden Leitungen sind:

28MHz, XCLK, XCLKEN, 7MHz, CCKQ, CCK und CDAC

Der Adreßbuspuffer

In der Abbildung 1.2.3 ist ein Puffer eingezeichnet, der den Adreßbus des Amiga mit dem Chip-RAM-Adreßbus und dem Registeradreßbus verbindet und die Prozessoradressen entsprechend multiplext. Dieser Puffer wurde komplett in Agnus integriert. Der Prozessoradreßbus kann jetzt direkt an die Leitungen A1 bis A18 von Fat Agnus angeschlossen werden. Mittels der beiden Signale RAMEN (RAM Enable) und RGEN (Register Enable) signalisiert der Adreßdecoder, daß der Prozessor auf das RAM oder den Registerbereich zugreifen will. Außerdem ist Agnus jetzt mit den Prozessorsignalen UDS, LDS und PR/W (Prozessor Read/Write) verbunden.

Die Steuerung des Chip-RAMs

Die Steuerung des Chip-RAMs wird jetzt komplett von Agnus übernommen. Agnus erzeugt die nötigen RAS- und CAS-Signale zusammen mit den gemultiplexten RAM-Adressen. Zusätzlich hat Agnus jetzt die Fähigkeit, weitere 512 KB-RAM zu verwalten, also insgesamt 1 Megabyte. Ausgewählt werden die beiden Banks mittels der RAM-Steuersignale: RAS0 und CAS0 für das Chip-RAM, RAS1 und CAS1 dagegen für das Erweiterungs-RAM.

An den entscheidenden Funktionen von Agnus, wie sie in Kapitel 1.2.3.1 beschrieben worden sind, hat sich nichts geändert.

Zusätzlich zu Fat-Agnus hat man einen vierten Customchip geschaffen. Dieser Chip hört auf den Namen Gary und übernimmt

die Funktion des Adreßdecoders und Bus-Controllers. Es erzeugt die Auswahlssignale für sämtliche Chips des Amiga, sowie VPA und DTACK für den Prozessor.

Nebenbei enthält Gary die Resetlogik und das Motor-Flip-Flop für das Diskettenlaufwerk (siehe 1.3.5).

1.3 Die Schnittstellen des Amiga

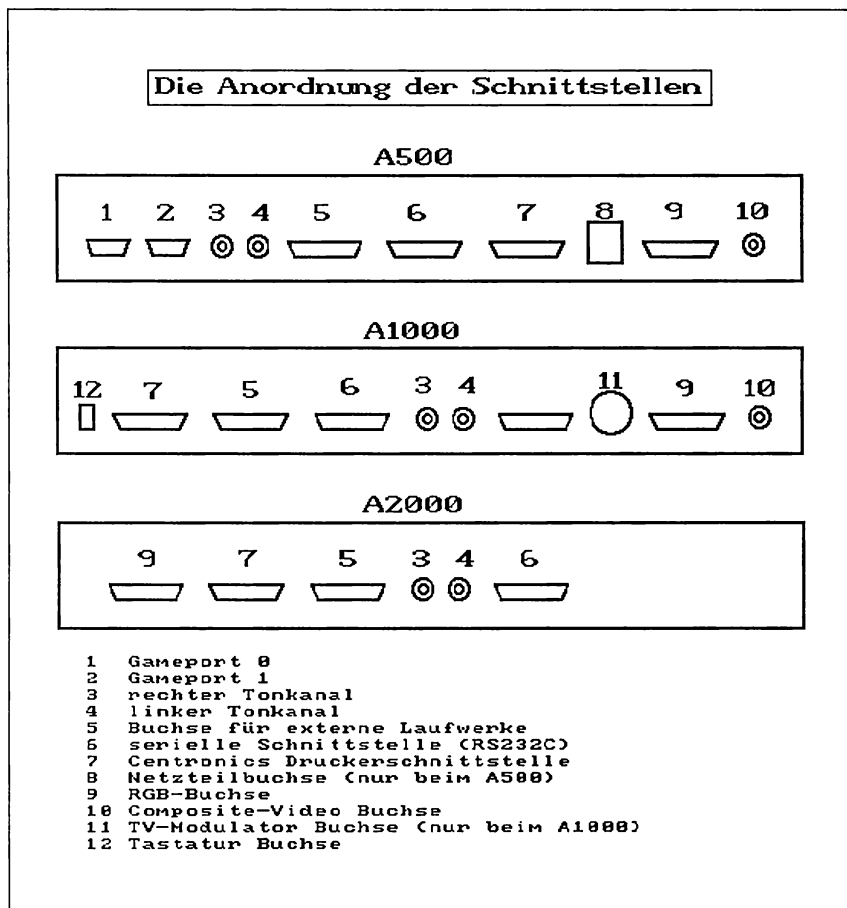


Abb. 1.3

1.3.1 Die Audio/Video-Schnittstellen

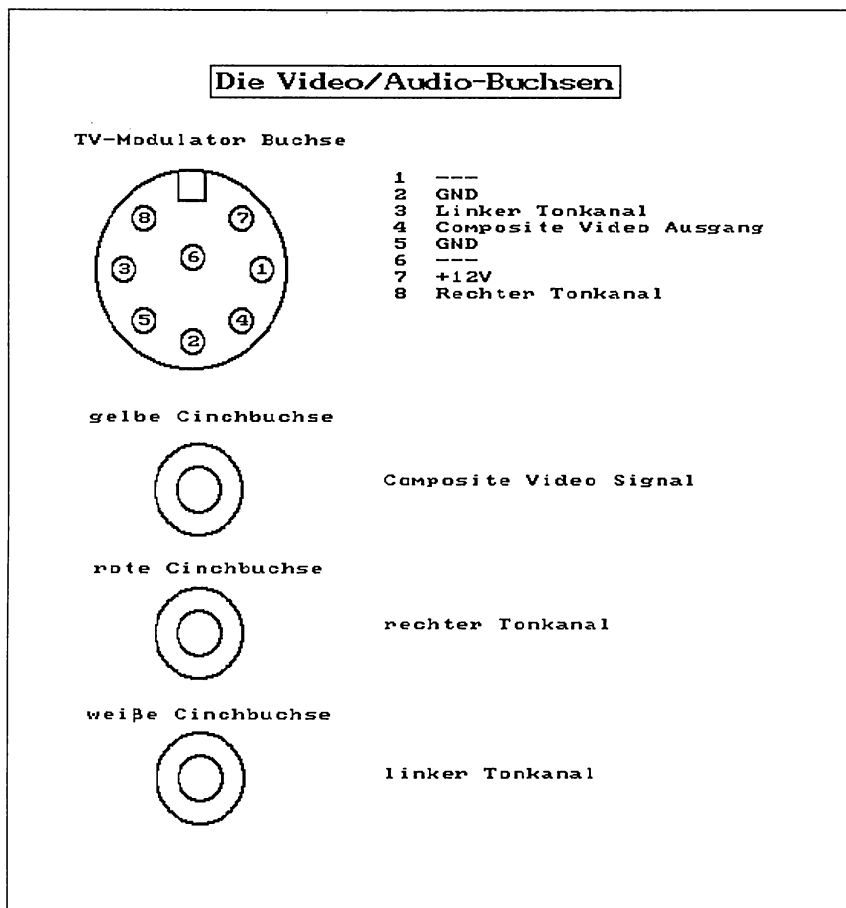


Abb. 1.3.1

Die Verteilung der Video-Buchsen ist bei den verschiedenen Amiga-Typen sehr unterschiedlich. Am spärlichsten ist der A2000 ausgestattet. Er verfügt über keinerlei Video-Ausgang, lediglich ein Connector zum Nachrüsten eines Video-Modulators oder eines Genlockinterfaces ist intern auf der Platine vorhanden. Sowohl der A500 als auch der A1000 verfügen über einen

Videoausgang in Form einer Cinch-Buchse. Das Videosignal an dieser Buchse ist beim A1000 ein Standard FBAS-Signal und mit allen handelsüblichen Videomonitoren zu verbinden. Beim A500 hat man allerdings gespart. Er liefert lediglich ein BAS-Signal, d.h. nur ein Schwarzweiß-Videosignal. Ein weiteres Problem sind die alten A1000-Typen, die noch nicht mit einer deutschen Tastatur ausgerüstet sind. Bei diesen Modellen liefert der Video-Ausgang ein NTSC-Signal, d.h. ein Videosignal nach amerikanischer Norm. Auf einem PAL-Farbmonitor (wie dem Amiga-Monitor) erscheint das Bild dann schwarzweiß mit senkrechten Streifen. Ein einwandfreies PAL-Farbbild liefern also nur die neuen A1000-Typen mit der deutschen Tastatur. Bei allen Modellen läuft das Videosignal über einen Transistorpuffer mit einem Ausgangslängswiderstand von 75 Ohm. Es ist damit absolut dauerkurzschlußfest.

Das Audiosignal liegt bei allen Amiga-Modellen an zwei Cinch-Buchsen an der Rückfront an. Der rechte Stereokanal liegt wie allgemein üblich an der roten Cinch-Buchse, der linke an der weißen. Mit einem handelsüblichen Stereo-Cinch-Kabel kann man diese Buchsen mit einer Stereoanlage (AUX-, TAPE- oder CD-Eingang) verbinden, was übrigens nur wärmstens zu empfehlen ist. Der Ausgangswiderstand jedes Kanals beträgt 1 KOhm (1000 Ohm). Die Ausgänge sind ebenfalls kurzschlußfest und intern schon mit je einem 360 Ohm Widerstand belastet.

Über eine weitere Audio/Video-Buchse verfügt der A1000. Die sogenannte TV-Mod-Buchse war ursprünglich für den Anschluß eines HF-Modulators vorgesehen, über den dann ein handelsüblicher Fernseher als Bildschirm für den Amiga Verwendung finden könnte. Allerdings wurde dieser HF-Modulator nie gebaut...

Nun, geblieben ist eine Buchse, an der sowohl das Videosignal als auch beide Tonkanäle liegen. Dazu gibt es noch einen 12-Volt-Anschluß, der zur Stromversorgung des Modulators vorgesehen war. Der Videoausgang an dieser Buchse verfügt über einen eigenen Transistorpuffer, ist also nicht einfach nur mit der Cinch-Videobuchse verbunden. Die beiden Audiopins haben ebenfalls eigene 1 KOhm Ausgangswiderstände. Da sie aber

nicht mit einem internen Belastungswiderstand versehen sind, ist ihr Pegel im unbelasteten Zustand etwa viermal so hoch wie der der Audio-Cinch-Buchsen.

Die TV-Mod-Buchse ist eine achtpolige DIN-Buchse. Passende Stecker für solche Buchsen sind leider schlecht zu bekommen. Vielleicht hilft es bei der Beschaffung, wenn man weiß, daß die TV-Mod-Buchse identisch mit der Video-Buchse des C64 ist.

Hier noch eine kleine Umbauanleitung für all diejenigen, die über einen alten A1000 mit NTSC-Videoausgang verfügen. Durch einen einfachen Eingriff läßt sich nämlich das Farbsignal unterbrechen, und der Videoausgang liefert dann ein normales BAS-Schwarzweiß-Videosignal. Damit kann man z.B. besonders lang nachleuchtende Monochrom-Monitore anschließen, die dann ein angenehmeres Arbeiten im Interlace-Modus ermöglichen. Dies gilt genauso für den neuen A1000, denn wenn der Videoausgang zum Anschluß eines Monochrommonitors benutzt wird, stört das PAL-Farbsignal genauso. Statt senkrechter Streifen hat man bei PAL eben ein grießähnliches Punktmuster.

Der Umbau besteht aus folgenden Schritten:

1. Öffnen Sie das Gehäuse (5 Schrauben an der Unterseite).
2. Entfernen Sie das Abschirmblech.
3. Suchen Sie direkt hinter der Videobuchse ein Chip mit der Bezeichnung MC 1377. Neben dem Chip ist auf der Platine auch noch die Nummer U7B aufgedruckt.
4. In der Nähe von Pin 10 dieses ICs befindet sich ein Kondensator von 1 Nanofarad (1 nF) mit der Bezeichnung C47 auf der Platine. Über diesen Kondensator gelang das Farbsignal an Pin 10 des ICs. Löten Sie den Kondensator aus oder knipsen Sie ihn mit einem Seitenschneider einfach ab.

5. Bauen Sie Ihren Amiga wieder zusammen und schließen Sie einen Monitor am Videoausgang an. Sie müßten jetzt ein einwandfreies Schwarzweißbild erhalten.

1.3.2 Die RGB-Buchse

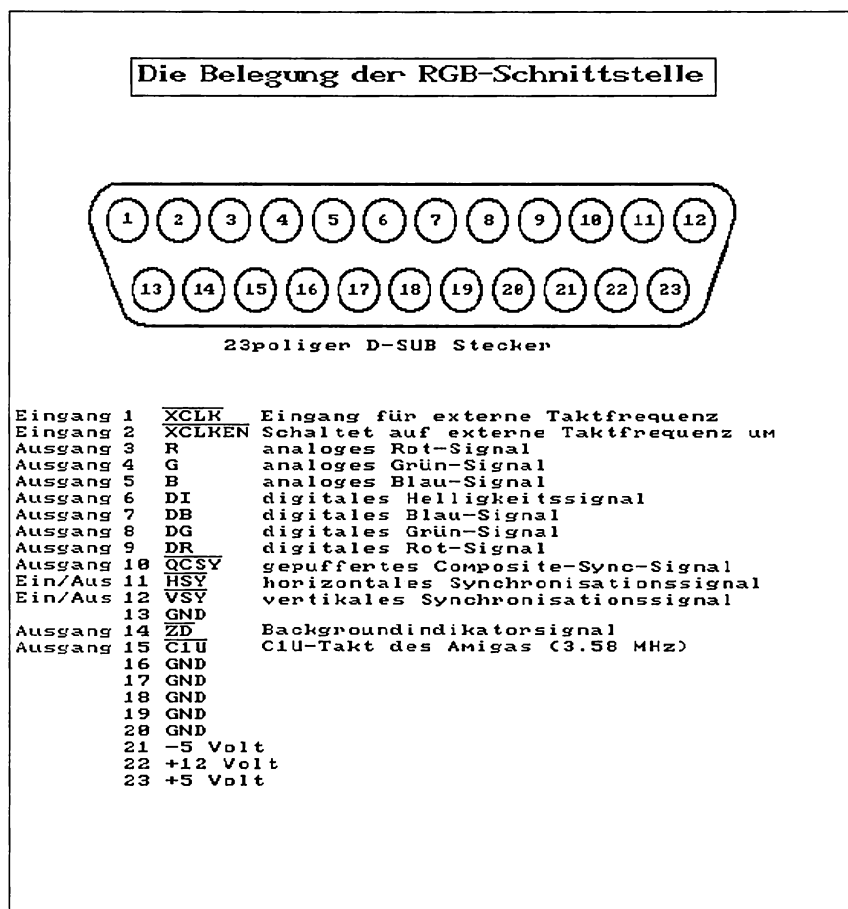


Abb. 1.3.2

Die RGB-Buchse ist bei allen drei Amiga-Modellen identisch. Sie ermöglicht den Anschluß der verschiedenen RGB-Monitore, aber auch spezieller Erweiterungen wie z.B. eines Genlock-Adapters. Zum Anschluß eines analogen RGB-Monitors wie z.B. des Amiga-Monitors dienen die drei analogen RGB-Ausgänge und der Composite-Sync-Ausgang. Das RGB-Signal an diesen drei Leitungen entsteht durch die Umwandlung der gepufferten, digitalen RGB-Signale von Denise in die entsprechenden Analogsignale mittels dreier 4-Bit-Digital-Analog-Wandler. Das Composite-Sync-Signal stammt von Agnus und wird durch Mischung des horizontalen und vertikalen Synchronsignals gebildet. Sämtliche dieser vier Leitungen sind mit Transistorpuffern und 75 Ohm Ausgangswiderständen versehen, wodurch sie unempfindlich gegen Kurzschlüsse werden.

Zum Anschluß eines digitalen RGB-Monitors sind die Leitungen DI, DB, DG und DR vorgesehen. Als Quelle der digitalen RGB-Signale dient das digitale RGB-Ausgangssignal von Denise (s. 1.2.3.2). Die drei Farbleitungen sind jeweils mit der höchstwertigen Farbleitung von Denise verbunden. DB z.B. mit B3 von Denise. Allerdings liegt zwischen Denise und den Ausgängen noch ein Puffer vom Typ 74HC244. Die Intensitäts- oder Helligkeitsleitung DI ist interessanterweise mit der B0 Leitung verbunden. Die vier Leitungen laufen über 47 Ohm Ausgangslängswiderstände und haben, da sie ja aus dem 74HC244 kommen, TTL-Pegel.

Für Monitore, die getrennte Synchronisationssignale benötigen, gibt es die HSY- und VSY-Anschlußpins an der RGB-Buchse. Bei diesen Leitungen ist etwas Vorsicht geboten, sie sind über 47 Ohm Widerstände direkt mit den HSY- und VSY-Pins von Agnus verbunden. Sie haben ebenfalls TTL-Pegel.

Wird das Genlock-Bit in Agnus gesetzt (siehe Kapitel über Programmierung der Hardware), dann werden diese beiden Leitungen zu Eingängen. Der Amiga synchronisiert dann sein eigenes Videosignal nach den Synchronisationssignalen auf der HSY- und VSY-Leitung. Auch wenn diese Leitungen als Eingang geschaltet sind, richten sie sich nach TTL-Pegel. Dabei sind die Synchronsignale wie allgemein üblich Low-aktiv, d.h. im Nor-

malzustand sind die Leitungen auf 5 Volt. Nur während des aktiven Synchronisationsimpulses liegt die Leitung auf 0 Volt.

Ein weiteres Signal, das ebenfalls mit Genlock in Verbindung steht, ist das ZD-Signal (Zero Detect). Der Amiga legt dieses Signal immer dann auf Low, wenn der gerade dargestellte Punkt ein Punkt des Bildschirmhintergrunds ist, in anderen Worten, wenn dessen Farbe aus Farbregister 0 stammt.

Während der vertikalen Austastlücke, also während $VS\bar{Y} = 0$ ist, ändert sich die Funktion der ZD-Leitung. Dann liegt an ihr der Wert des GAUD-Bits (Genlock Audio Enable) aus dem Agnus Register \$100 (BPLCON0). Dieses Signal wird vom Genlock-Interface zum Schalten des Tonsignals benutzt.

Normalerweise ist die ZD-Leitung für einen normalen Anwender uninteressant, sie wird nur von Genlock-Interfaces benötigt. Das ZD-Signal von Denise Pin 33 läuft auch über einen 74HC244 Treiber. Der Pegel liegt dabei auf TTL-Niveau.

Die restlichen Leitungen der RGB-Buchse haben nichts mehr mit dem RGB-Signal zu tun.

Die CIU-Leitung ist eine 3.58 MHz Taktleitung und entspricht dem invertierten CLK-Signal der Customchips.

Die XCLK (External Clock) und XCLKEN (External Clock Enable) dienen zum Einspeisen einer externen Taktfrequenz in den Amiga. Sämtliche Taktsignale im Amiga werden von einem einzigen 28-MHz-Takt abgeleitet. Dieser 28-MHz-Mastertakt kann durch eine andere Taktfrequenz an dem XCLK-Eingang ersetzt werden, indem man die XCLKEN-Leitung auf 0 legt. Dadurch kann der Amiga z.B. beschleunigt werden, wenn man einen 32-MHz- oder sogar noch höheren Takt an XCLK legt. Wieweit die Amiga-Hardware bei diesen Frequenzen noch funktionsfähig bleibt, müßte experimentell ermittelt werden. Bei Verwendung der XCLK- und XCLKEN-Leitungen sollte der Groundpin 13 verwendet werden. Er ist direkt mit der Masseleitung der Takterzeugung verbunden.

1.3.3 Die Centronics-Schnittstelle

Die Pinbelegung der Centronicsstecker

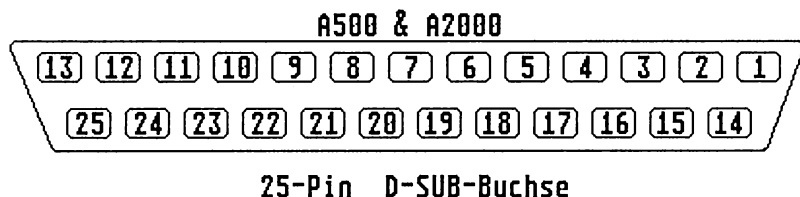
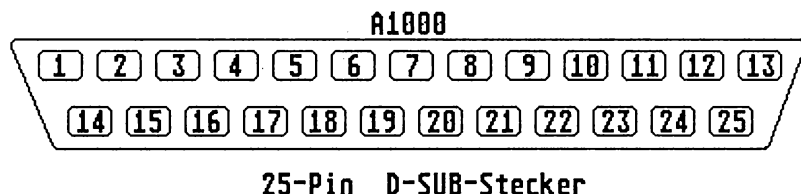


Abb. 1.3.3

Ausgang	1	/Strobe - Daten bereit
Ein/Aus	2	Datenbit 0
Ein/Aus	3	Datenbit 1
Ein/Aus	4	Datenbit 2
Ein/Aus	5	Datenbit 3
Ein/Aus	6	Datenbit 4
Ein/Aus	7	Datenbit 5
Ein/Aus	8	Datenbit 6
Ein/Aus	9	Datenbit 7
Eingang	10	/Acknowledge - Datenübernahmesignal
Ein/Aus	11	BUSY - Drucker beschäftigt
Ein/Aus	12	Paper Out - Papierende erreicht

Ein/Aus	13	Select - Drucker ON-LINE
	14	+5 Volt
	15	unbelegt
Ausgang	16	Reset / gepufferte Resetleitung vom Amiga
	17-25	GND

Beim A1000 sind einige Leitungen anders belegt:

	14-22	GND
	23	+5 Volt
	24	unbelegt
Ausgang	25	Reset / gepufferte Resetleitung vom Amiga

Die Centronics-Schnittstelle des Amiga erfreut das Herz jedes Computerfreaks. Sie ist vollkommen PC-kompatibel. Jeder IBM-kompatible Drucker kann direkt an ihr angeschlossen werden. Damit steht dem Amiga ein riesiges Reservoir an anschlußfertigen Druckern zur Verfügung. Allerdings gilt dies leider nur für den A500 und den A2000. Beim A1000 stimmt der Centronicsport nicht mit dem PC-Standard überein. Erstens wurde statt der üblichen DSUB-Buchse ein Stecker verwendet und zweitens liegt an Pin 23 eine Spannung von 5 Volt an. Diese Leitung ist bei handelsüblichen Druckerkabeln oft mit Masse (GND) verbunden. Steckt man ein solches Kabel beim A1000 ein, entsteht ein Kurzschluß, der ernsthafte Beschädigungen des Amiga zur Folge haben kann. Man ist beim A1000 meistens gezwungen, selber zum Lötkolben zu greifen und ein entsprechendes Kabel zu basteln.

Intern sind sämtliche Centronics-Port-Leitungen (bis auf 5 Volt und Reset) direkt mit den Port-Leitungen der einzelnen CIAs verbunden. Die genaue Zuordnung ist folgende:

Centronics Pin Nr.	Funktion	CIA	Pin	Pinbezeichnung
1	Strobe	A	18	PC
2	Datenbit 0	A	10	PB0
3	Datenbit 1	A	11	PB1
4	Datenbit 2	A	12	PB2
5	Datenbit 3	A	13	PB3
6	Datenbit 4	A	14	PB4
7	Datenbit 5	A	15	PB5
8	Datenbit 6	A	16	PB6
9	Datenbit 7	A	17	PB7
10	Acknowledge	A	24	PB8
11	Busy	B	2 und 39	PA0 SP
12	Paper Out	B	3 und 40	PA1 CNT
13	Select	B	4	PA2

Die Centronics-Schnittstelle ist eine Parallelschnittstelle. Das Datenbyte liegt an den acht Datenleitungen an. Hat der Computer ein gültiges Byte auf die Datenleitungen gelegt, setzt er die STROBE-Leitung für 1,4 Microsekunden auf 0 und signalisiert damit dem Drucker, das jetzt ein gültiges Byte bereitsteht. Der Drucker muß dies quittieren, indem er die Acknowledge-Leitung mindestens eine Mikrosekunde lang auf 0 legt. Erst nachdem der Drucker damit gezeigt hat, daß er das Datenbyte übernommen hat, legt der Computer das nächste Byte auf den Bus.

Mit der Busy-Leitung signalisiert der Drucker, daß er gerade beschäftigt ist und daher keine weiteren Daten annehmen kann. Dies tritt z.B. ein, wenn der Druckpuffer voll ist. Der Computer wartet dann, bis BUSY wieder high wird, bevor er die Übertragung fortsetzt. Mit der Paper-Out-Leitung teilt der Drucker mit, daß der Papiervorrat zu Ende ist. (Der Name sagt ja schon alles...) Die Select-Leitung wird ebenfalls vom Drucker gesteuert. Sie zeigt an, ob er ON-LINE (Selected, SEL auf High) oder OFF-LINE (Unselected, SEL auf Low) ist.

Der Centronics-Port eignet sich auch hervorragend als Universalschnittstelle zum Anschluß selbstgebastelter Erweiterungen

wie z.B. eines Audiodigitalisierers oder EPROM-Brenners, da sich fast alle Leitungen beliebig als Ein- oder Ausgang programmieren lassen.

1.3.4 Die serielle Schnittstelle

Die Pinbelegung der RS232 Buchse

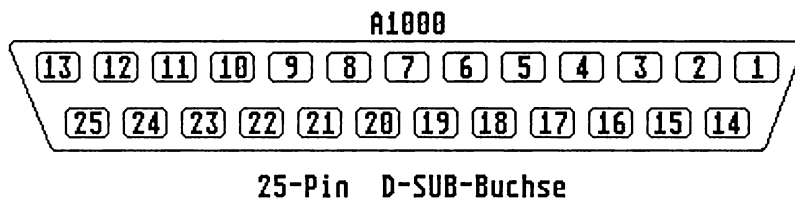
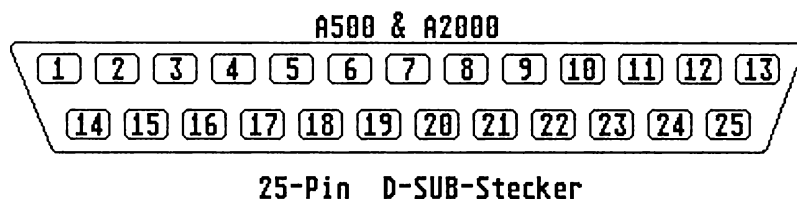


Abb. 1.3.4

	1	GND	Abschirmung Masse (Frame Ground)
Ausgang	2	TXD	Transmit Data
Eingang	3	RXD	Receive Data
Ausgang	4	RTS	Request To Send
Eingang	5	CTS	Clear To Send
Eingang	6	DSR	Data Set Ready
	7	GND	Signal Masse
Eingang	8	CD	Carrier Detect
	9	+12 Volt	
	10	-12 Volt	

Ausgang	11	AUDOUT	Ausgang linker Audiokanal
	12	unbenutzt	
	13	unbenutzt	
	14	unbenutzt	
	15	unbenutzt	
	16	unbenutzt	
	17	unbenutzt	
Eingang	18	AUDIN	Eingang rechter Audiokanal
	19	unbenutzt	
Ausgang	20	DTR	Data Terminal Ready
	21	unbenutzt	
Eingang	22	RI	Ring Indicator
	23	unbenutzt	
	24	unbenutzt	
	25	unbenutzt	

Beim A1000 sind wieder einige Leitungen anders belegt:

	9	unbenutzt	
	10	unbenutzt	
	11	unbenutzt	
	12	unbenutzt	
	13	unbenutzt	
	14	-5 Volt	
Ausgang	15	AUDOUT	Ausgang linker Audiokanal
Eingang	16	AUDIN	Eingang rechter Audiokanal
Ausgang	17	E	gepuffertes E-Takt (716 KHz)
Eingang	18	/INT2	Interrupt der Ebene 2 über Paula
	19	unbenutzt	
Ausgang	20	DTR	Data Terminal Ready
	21	+5 Volt	
	22	unbenutzt	
	23	+12 Volt	
Ausgang	24	MCLK	Taktausgang 3.58 MHz
Ausgang	25	/MRES	gepuffertes Reset-Ausgang

Die serielle Schnittstelle besitzt alle üblichen RS232-Signalleitungen. Zusätzlich liegen noch viele Signale an dieser Schnittstelle, die nichts mit der seriellen Kommunikation zu tun haben.

Leider weicht die Belegung der Buchse beim A500 und A1000 wieder voneinander ab.

Zur RS232 gehören die Leitungen TXD, RXD, DSR, CTS, DTR, RTS und CD. Die Leitungen TXD und RXD sind die eigentlichen seriellen Datenleitungen. Die TXD-Leitung ist der serielle Ausgang des Amiga, RXD der Eingang. Sie sind mit den entsprechenden Leitungen Paulas verbunden. Die DTR-Leitung zeigt dem angeschlossenen Peripheriegerät an, das die serielle Schnittstelle des Amiga in Betrieb ist. Die DSR-Leitung ist das Gegenstück dazu, mit ihr signalisiert das Peripheriegerät dem Amiga, daß seine Schnittstelle betriebsbereit ist. Die RTS-Leitung zeigt dem Peripheriegerät an, daß der Amiga jetzt serielle Daten über die RS232 senden will. Mit der CTS-Leitung signalisiert dieses dann, daß es jetzt empfangsbereit ist. Das CD-Signal wird gewöhnlich nur von einem Modem benutzt. Mit ihm zeigt dieses an, daß es eine Trägerfrequenz empfängt.

Diese fünf RS232-Steuerleitungen sind mit CIAB, PA3 - PA7 folgendermaßen verbunden: DSR-PA3; CTS-PA4; CD-PA5; RTS-PA6; DTR-PA7. Die RI-Leitung ist über einen Transistor mit der SEL-Leitung der Centronicsschnittstelle verbunden.

Erfreulicherweise hat man die RS232-Leitungen nicht direkt mit den Chips verbunden, sondern über RS232-Treiber geführt. Damit kann man diese Schnittstelle über ein entsprechendes Kabel mit fast allen gängigen Terminals oder Modems verbinden. Als Ausgangstreiber wurden invertierende RS232-Pegelwandler vom Typ 1488 verwendet. Sie werden mit Versorgungsspannungen von +12 und -5 Volt betrieben. In diesem Bereich bewegt sich daher auch der Ausgangspegel. Als Eingangspuffer wurden Chips vom Typ 1489A benutzt. Damit akzeptieren die Eingänge alle Spannungen zwischen -12 und +0.5 Volt als Low und den Bereich von +3 bis +25 Volt als High.

Die Konventionen für RS232-Schnittstellen besagen, daß die Steuerleitungen aktiv high sind, während im Gegensatz dazu bei den Datenleitungen RXD und TXD eine logische 1 durch einen negativen Pegel dargestellt wird. Da die Treiber invertieren, sind auch die entsprechenden Portbits in CIAB lowaktiv. D.h. ein Bit

mit dem Wert 0 in CIAB setzt die entsprechende RS232-Steuerung auf High. Gleiches gilt natürlich auch für die Eingänge.

Die übrigen Leitungen der RS232-Schnittstelle haben nichts mit RS232 zu tun. Die AUDOUT-Leitung ist mit dem linken Tonkanal verbunden und mit einem eigenen 1 KOhm Ausgangslängswiderstand versehen. Die AUDIN-Leitung ist über einen 47 Ohm Widerstand direkt mit dem AUDR-Pin von Paula verbunden. Audiosignale, die auf der AUDIN-Leitung in den Amiga eingespeist werden, gelangen zusammen mit dem rechten Tonkanal von Paula über den Tiefpaßfilter (s. Audio-Programmierung) zum rechten Audioausgang. Sonst geschieht nichts mit dem Signal. Die INT2-Leitung ist direkt mit dem INT2-Eingang von Paula verbunden und kann einen Prozessor-Interrupt der Ebene 2 auslösen, wenn das entsprechende Maskenbit in Paula gesetzt ist (s. Kapitel über Interrupts). Die E-Leitung ist über einen Puffer mit dem E-Takt des Prozessors verbunden (s. 1.2.1). An der MCLK-Leitung liegt eine Frequenz von 3.58 MHz. Allerdings ist dieser 3.58-MHz-Takt in seiner Phasenlage weder mit dem Takt an der RGB-Schnittstelle noch mit den beiden 3.58-MHz-Frequenzen der Customchips identisch. Zu guter Letzt liegt auch noch das Resetsignal an dieser Schnittstelle, selbstverständlich gepuffert, wie man es vom Amiga gewohnt ist.

1.3.5 Die Anschlußbuchse für externe Diskettenlaufwerke

Die Pinbelegung der Anschlußbuchse für externe Diskettenlaufwerke

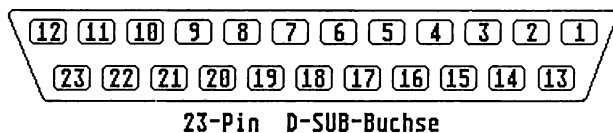


Abb. 1.3.5.1

Eingang	1	/RDY Diskbereitschaftssignal
Eingang	2	/DKRD Lesedaten von Diskette
	3	GND
	4	GND
	5	GND
	6	GND
	7	GND
Ausgang	8	/MTRX Motor An/Aus
Ausgang	9	/SEL2 Selektierung von Laufwerk 2
Ausgang	10	/DRES Floppyreset (Abschalten der Motoren)
Eingang	11	/CHNG Diskettenwechsel
	12	+5 Volt
Ausgang	13	/SIDE Seitenauswahl
Eingang	14	/WPRO Write-Protect-Leitung
Eingang	15	/TK0 Spur 0 Indikator
Ausgang	16	/DKWE Umschalten auf Schreiben
Ausgang	17	/DKWD Schreibdaten zur Diskette
Ausgang	18	/STEP Bewegung des Schreib/Lesekopfs
Ausgang	19	/DIR Richtung der Kopfbewegung
Ausgang	20	/SEL3 Selektierung von Laufwerk 3
Ausgang	21	/SEL1 Selektierung von Laufwerk 1
Eingang	22	/INDEX INDEX-Signal des Laufwerks
	23	+12 Volt

Die Pinbelegung des internen Floppystecker

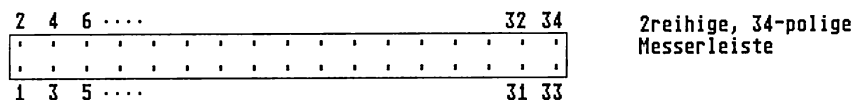


Abb. 1.3.5.2

Alle ungeraden Pins liegen auf GND.

2	/CHNG	4	/INUSE (verbunden mit /MTR0D)
6	unbenutzt	8	/INDEX
10	/SEL0	12	unbenutzt
14	unbenutzt	16	/MTR0
18	DIR	20	/STEP

22	/DKWD	24	/DKWE
26	/TK0	28	/WPRO
30	/DKRD	32	/SIDE
34	/RDY		

Stromversorgungsbuchse für das interne Laufwerk:

- 1 +5 Volt
- 2 GND
- 3 GND
- 4 +12 Volt

Der Floppy-Anschluß des Amiga ist kompatibel zu dem Shugart-Bus. Er ermöglicht den Anschluß von bis zu vier Shugart-Bus-kompatiblen Laufwerken. Die Auswahl der Laufwerke erfolgt über vier Auswahlleitungen, die sogenannten DriveSelectX-Signale (SELX), wobei X für die Nummer des Laufwerks steht, das durch diese Leitung angewählt werden kann. Da der Amiga schon über eine eingebaute Floppy verfügt, liegen am Anschluß für externe Laufwerke nur noch die SEL1-, SEL2- und SEL3-Leitungen. Die SEL0-Leitung ist mit dem internen Connector verbunden, an dem das eingebaute Laufwerk angeschlossen ist. Es folgen nun die Funktionen der einzelnen Shugart-Bus-Signale beim Amiga:

SELX

Mit der SELX-Leitung wählt der Amiga eines der vier Laufwerke aus. Außer der MTRX- und der DRES-Leitung sind alle anderen Signale nur dann aktiv, wenn das Laufwerk mit der entsprechenden SELX-Leitung aktiviert wurde.

MTRX

Normalerweise bewirkt diese Leitung, daß alle angeschlossenen Laufwerke ihren Motor einschalten. Bei maximal vier möglichen Laufwerken ist dies keine befriedigende Lösung. Deshalb hat man beim Amiga für jedes Laufwerk ein Flip-Flop vorgesehen (Ein Flip-Flop ist ein elektronischer Baustein, der in der Lage ist, ein Datenbit zu speichern). Immer wenn die SEL-Leitung

des betreffenden Laufwerks auf Low geht, übernimmt das Flip-Flop dieses Laufwerks den Wert der MTRX-Leitung. Der Ausgang des Flip-Flops ist mit der MTR-Leitung des Laufwerks verbunden. Auf diese Weise können die Motoren der Laufwerke unabhängig voneinander ein- bzw. ausgeschaltet werden. Legt man z.B. die SEL0-Leitung auf Low, während sich die MTRX-Leitung auf 0 befindet, läuft der Motor des eingebauten Laufwerks an. Dieses Flip-Flop befindet sich für das eingebaute Laufwerk schon auf der Hauptplatine. Für jedes externe Laufwerk wird ein weiteres benötigt. Bei der 1010-Zweifloppy von Commodore hat man es auf einer kleinen Adapterplatine untergebracht.

RDY

Wenn die MTR-Leitung des entsprechenden Laufwerks auf 0 ist, signalisiert die RDY-Leitung (Ready) dem Amiga, daß der Laufwerksmotor seine Nenndrehzahl erreicht hat und das Laufwerk jetzt bereit für Schreib- oder Lesezugriffe ist. Ist die MTR-Leitung dagegen auf 1, der Motor des Laufwerks also abgeschaltet, dient sie einem speziellen Identifikationsmodus. (s.u.)

DRES

Die DRES-Leitung (Drive Reset) ist mit der normalen Resetleitung des Amiga verbunden und setzt lediglich oben erwähnte Motor-Flip-Flops zurück, d.h. alle Laufwerksmotoren werden abgeschaltet.

DKRD

Über die DKRD-Leitung (Disk Read Data) gelangen die Daten des durch SELX ausgewählten Laufwerks in den Amiga, und zwar zum DKRD-Pin von Paula.

DKWD (Disk Write Data)

Daten von Paulas DKWD-Pin zum aktuellen Laufwerk, das sie dann auf die Diskette schreibt.

DKWE

Die DKWE-Leitung (DiskWriteEnable) schaltet das Laufwerk von Lesen auf Schreiben um. Ist die Leitung high, werden die Daten von der Diskette gelesen, liegt sie dagegen auf Low, kann man Daten auf die Disk schreiben.

SIDE

Die SIDE-Leitung wählt aus, auf welcher Diskettenseite die Daten gelesen bzw. geschrieben werden. Ist sie high, ist Seite 0, d.h. der untere Schreib/Lesekopf, aktiv. Liegt sie auf Low, ist Seite 1 angewählt.

WPRO

Die WPRO-Leitung (Write Protect) teilt dem Amiga mit, ob die eingelegte Diskette schreibgeschützt ist. Liegt eine schreibgeschützte Diskette im Laufwerk, ist die WPRO-Leitung auf 0.

STEP

Eine positive Flanke an der STEP-Leitung (Wechsel von Low nach High) bewegt den Schreib/Lesekopf des Laufwerks um eine Spur nach innen oder außen, je nach dem Zustand der DIR-Leitung. Bevor die SEL-Leitung des aktivierten Laufwerks wieder auf High zurückgesetzt wird, sollte das STEP-Signal auf jeden Fall auf 1 liegen, da es sonst Probleme mit der Diskwechsel-Erkennung geben kann.

DIR

Die DIR-Leitung (Direction) legt die Richtung fest, in die der Kopf bei einem Impuls an der STEP-Leitung bewegt wird. Low bedeutet in Richtung Diskettenmitte und High nach außen in Richtung Diskettenrand. Spur 0 ist die äußerste Spur der Diskette.

TK0

Die TK0-Leitung (Track 0) ist immer dann auf Low, wenn sich der Schreib/Lesekopf des aktivierten Laufwerks auf Spur 0 befindet. Damit besteht die Möglichkeit, den Kopf in eine definierte Position zu bringen.

INDEX

Das INDEX-Signal ist ein kurzer Impuls, den das Laufwerk einmal pro Diskettenumdrehung liefert, und zwar immer zwischen Anfang und Ende einer Spur.

CHNG

Mit der CHNG-Leitung (Change) signalisiert das Laufwerk dem Amiga einen Diskettenwechsel. Sobald die Diskette aus dem Laufwerk genommen wird, legt es die CHNG-Leitung auf 0. Die Leitung bleibt auf 0, bis der Computer einen STEP-Impuls auslöst. Ist zu diesem Zeitpunkt schon wieder eine Diskette im Laufwerk, springt CHNG auf 1 zurück. Ansonsten bleibt sie auf 0, und der Computer muß weiterhin in regelmäßigen Abständen einen STEP-Impuls auslösen, um zu erkennen, ob sich wieder eine Diskette im Laufwerk befindet. Diese regelmäßigen STEP-Impulse sind die Ursache der Knacklaute, die das Amiga-Laufwerk von sich gibt, wenn keine Diskette eingelegt ist.

INUSE

Die INUSE-Leitung existiert nur am internen Floppystecker. Legt man diese Leitung auf 0, schaltet das Laufwerk seine Leuchtdiode ein. Normalerweise verbindet man diese Leitung mit der MTR-Leitung.

Um zu erkennen, ob ein Laufwerk am Bus angeschlossen ist, gibt es einen speziellen Identifikationsmodus. Dabei wird ein serielles, 32-Bit langes Datenwort von dem Laufwerk gelesen. Um diese Identifikation zu starten, muß die MTR-Leitung des betreffenden Laufwerks kurz ein- und darauf wieder ausgeschaltet werden (Wie dies geschieht, steht bei der Beschreibung des MTRX-Signals). Dadurch wird im Laufwerk das serielle

Schieberegister zurückgesetzt. Danach kann man die einzelnen Datenbits lesen, indem man die SELX-Leitung auf Low legt, den Wert der RDY-Leitung als Datenbit liest und danach die SELX-Leitung wieder auf High legt. Diesen Vorgang wiederholt man 32mal. Das zuerst empfangene Bit ist das MSB (Most Significant Bit / höchstwertiges Bit) des Datenworts. Da die RDY-Leitung Low-aktiv ist, müssen die Datenbits invertiert werden.

Folgende festgelegte Definitionen für externe Laufwerke gibt es:

\$0000 0000	Kein Laufwerk angeschlossen (00)
\$FFFF FFFF	Standard Amiga-3 $\frac{1}{2}$ -Zoll-Laufwerk (11)
\$5555 5555	Amiga-5 $\frac{1}{4}$ -Zoll-Laufwerk, 2*40 Spuren (01)

Wie man sieht, gibt es derzeit sowenig verschiedene Identifikationen, daß es völlig reicht, die ersten beiden Bits zu lesen. Die Werte in Klammern geben die Kombinationen dieser beiden Bits an.

Wie schon oben erwähnt, haben alle Leitungen außer DRES nur eine Wirkung auf das durch SELX ausgewählte Laufwerk. Ursprünglich wirkte auch die MTRX-Leitung unabhängig von SELX, aber die Amiga-Entwickler haben dies durch Hinzufügen erwähnter Motor-Flip-Flops geändert.

Alle Leitungen des Shugart-Bus sind low-aktiv, da die Ausgänge sowohl im Amiga als auch in den Laufwerken mit Open-Collector-Treibern versehen sind. Im Amiga sind diese Treiber vom Typ 7407.

Die vier Eingänge CHNG, WPRO, TK0 und RDY sind in dieser Reihenfolge direkt mit PA4 - PA7 von CIA-A verbunden. Die acht Ausgänge STEP, DIR, SIDE, SEL0, SEL1, SEL2, SEL3, MTR kommen von CIA-B, PB0-7 und sind über oben erwähnte Treiber mit dem internen und externen Floppyconnector verbunden. Da diese Treiber nicht invertieren, sind die Bits in den CIAs ebenfalls invertiert. Die DKRD-, DKWD- und DKWE-Leitungen kommen von Paula.

Außer der MTRX-Leitung und den SEL-Signalen sind die Anschlüsse am internen und externen Floppyanschluß identisch. Das interne Laufwerk ist mit SEL0 verbunden. Seine MTR-Leitung läuft über das oben beschriebene Motor-Flip-Flop.

Am externen Anschluß liegt direkt die MTRX-Leitung vom CIA und die übrigen drei SEL-Leitungen.

Anschluß externer Laufwerke an den Amiga

Beim Amiga kann man kaum längere Zeit mit einem einzigen Laufwerk auskommen. Wenn also der Wunsch nach einem Zweitlaufwerk übermächtig geworden ist, stellt sich die Frage: kaufen oder selberbauen? Da normale beidseitige 3½-Zoll-Laufwerke, wie sie im Amiga verwendet werden, in letzter Zeit für einen Bruchteil des Preises des original Amiga Zweitlaufwerks A1010 angeboten werden, ist der Selbstbau empfehlenswert. Was ist zu tun?

Der Anschlußstecker eines 3½ Zoll Standardlaufwerks, wie z.B. eines NEC FD1035 oder FD1036, ist identisch mit dem 34-poligen Anschluß für das interne Laufwerk im Amiga, ebenso der Stromversorgungsstecker. Um ein Laufwerk wie die FD1035 an den Amiga anzuschließen, muß lediglich noch das Motor-Flip-Flop hinzugefügt werden. Abbildung 1.3.5.3 zeigt die entsprechende Schaltung.

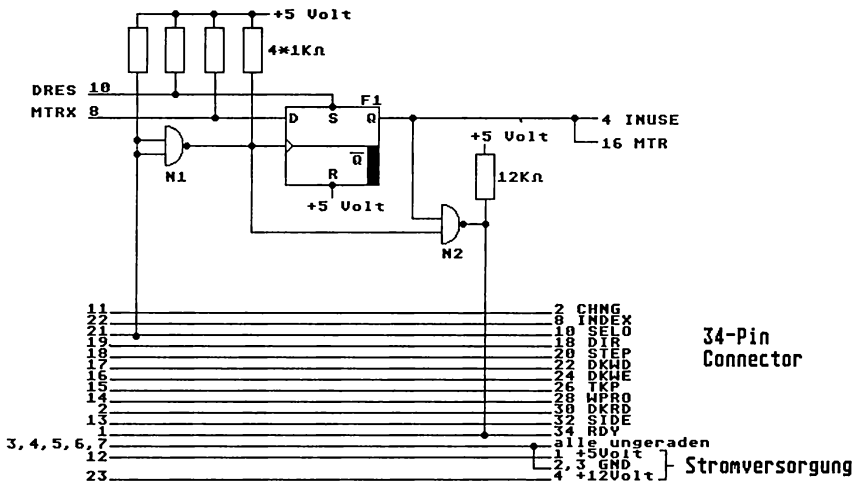


Abb. 1.3.5.3

Wie man sieht, speichert das mit F1 bezeichnete Flip-Flop das Signal an der MTRX-Leitung, wenn die SEL1-Leitung von High auf Low geht. Da das Flip-Flop den Wert an seinem Dateneingang mit der positiven Flanke des Taktsignals speichert, muß SEL1 invertiert werden. Dies erledigt das Nandgatter N1. Der Q-Ausgang ist direkt mit dem MTR-Eingang des Zweitlaufwerks verbunden.

Das mit N2 bezeichnete Nandgatter hat nichts mit der Motorsteuerung zu tun. Es dient dem oben erwähnten Identifikationsmodus, der von den meisten handelsüblichen Laufwerken nicht unterstützt wird. Immer wenn der Motor abgeschaltet ist und die SEL1-Leitung aktiv, also auf 0, legt das Gatter die RDY-Leitung auf Low. Dadurch erkennt der Amiga das Zweitlaufwerk als 3½ Zoll Standardlaufwerk mit der Nummer DF1: an.

Da von beiden verwendeten ICs jeweils nur die Hälfte benötigt wird, reichen sie auch noch für ein zweites Zusatzlaufwerk aus. Die Eingänge von N1 müssen dann allerdings mit SEL2 verbunden werden (Pin 9 am externen Floppystecker).

Damit die CHNG-Leitung einwandfrei funktioniert, müssen bei manchen Laufwerken einige Jumper umgesteckt werden. Näheres dazu entnimmt man am besten dem Handbuch des Laufwerks. Als Beispiel sei die FD1035 von NEC genannt, bei ihr muß man den mit J1 bezeichneten Jumper kurzschließen.

1.3.6 Die Gameports

Die Pinbelegung der Gameports

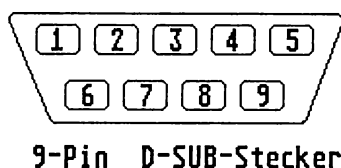


Abb. 1.3.6

Verwendung als:

	Mausport	Joystick	Paddle	Lightpen
Eingang	1 V-Impulse	Oben	unbenutzt	unbenutzt
Eingang	2 H-Impulse	Unten	unbenutzt	unbenutzt
Eingang	3 VQ-Impulse	Links	Linker Knopf	unbenutzt
Eingang	4 HQ-Impulse	Rechts	Rechter Knopf	unbenutzt
Ein/Aus	5 (Knopf 3)	unbenutzt	Rechtes Poti	Knopf
Ein/Aus	6 Knopf 1	Feuerknopf	unbenutzt	LP-Sigal
	7 +5 Volt	+5 Volt	+5 Volt	+5 Volt
	8 GND	GND	GND	GND
Ein/Aus	9 Knopf 2	unbenutzt	Linkes Poti	unbenutzt

Die Gameports sind Eingänge für die neben der Tastatur üblichen Eingabegeräte wie Maus, Joystick, Trackball, Paddle oder Lightpen. Es gibt zwei Gameports, der linke wird mit Gameport 0 und der rechte mit Gameport 1 bezeichnet. Die Belegung beider Gameports ist identisch. Lediglich die LP-Leitung ist nur bei Gameport 0 vorhanden. Intern sind die Gameports mit CIA-

A, Agnus, Denise und Paula verbunden. Im einzelnen sind die Pins folgendermaßen verschaltet:

Gameport 0:

Leitung Nr.	Chip	Pin
1	Denise	M0V (über Multiplexer)
2	Denise	M0H (über Multiplexer)
3	Denise	M1V (über Multiplexer)
4	Denise	M1H (über Multiplexer)
5	Paula	P0Y
6	CIA-A	PA6
	sowie Agnus	LP
9	Paula	P0X

Gameport 1:

1	Denise	M0V (über Multiplexer)
2	Denise	M0H (über Multiplexer)
3	Denise	M1V (über Multiplexer)
4	Denise	M1H (über Multiplexer)
5	Paula	P1Y
6	CIA-A	PA7
9	Paula	P1X

Die Funktion des oben erwähnten Multiplexers ist in Kapitel 1.2.3.2 schon erläutert worden.

Die Belegungen für die verschiedenen Eingabegeräte wurde so gewählt, daß fast alle handelsüblichen Joysticks, Mäuse, Paddles und Lightpens verwendet werden können. Es ist also möglich, Lightpens z.B. vom C64 weiterzuverwenden. Dabei ist die mit "Knopf" bezeichnete Leitung meist ein Schalter, der gedrückt wird, wenn man mit dem Lightpen den Schirm berührt. Die LP-Leitung ist das eigentliche Lightpensignal, das von der Elektronik des Griffels erzeugt wird, wenn der Elektronenstrahl an seiner Spitze vorbeiläuft.

Alle mit "Knopf" bezeichneten Leitungen und die vier Richtungen bei Joystickbetrieb sind Low-aktiv. In den verschiedenen

Eingabegeräten befinden sich Schalter, die mit dem entsprechenden Eingang und Masse (GND) verbunden sind. Ein High-Signal am Eingang bedeutet offener Schalter, ein geschlossener Schalter ruft dagegen ein Low hervor.

An die mit P0X, P0Y, P1X und P1Y bezeichneten Analogeingänge können veränderliche Widerstände, sog. Potentiometer, angeschlossen werden. Ihr Wert sollte 470 KOhm betragen. Sie werden zwischen +5 Volt und den entsprechenden Eingang geschaltet.

Die beiden Feuerknopf-Leitungen, die mit dem CIA-A verbunden sind, können natürlich auch als Ausgang programmiert werden. Allerdings muß man bei einem Schreibzugriff auf das Portregister aufpassen, daß man das unterste Portbit nicht überschreibt, da dies einen Systemzusammenbruch zur Folge hat (PA0:OVL).

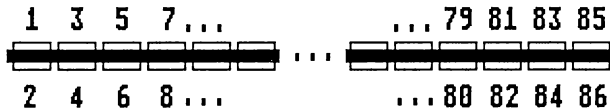
Die genaue Abfrage der Gameport-Leitungen wird im Kapitel über die Programmierung der Customchips erläutert.

Die +5 Volt Spannung an den beiden Gameports ist nicht direkt mit der Betriebsspannung des Amiga verbunden. Man hat eine Strombegrenzungsschaltung zwischengeschaltet. Sie begrenzt den kurzzeitigen Spitzenstrom auf 700 mA und den Dauerkurzschlußstrom auf 400 mA. Das macht diesen Ausgang absolut kurzschlußfest. Damit die Spannung an den beiden +5-Volt-Pins nicht zu stark abfällt, sollte die Stromentnahme an beiden Ports zusammen 250 mA nicht übersteigen.

Leider hat man diese sinnvolle Schutzschaltung beim A500 und A2000 eingespart.

1.3.7 Der Expansion-Port

Die Pinbelegung des Expansionport



86-poliger Platinenstecker A500 & A1000

Abb. 1.3.7

1	GND	2	GND
3	GND	4	GND
5	+5 Volt	6	+5 Volt
7	Erweiterung	8	-5 Volt
9	Erw. (28M beim A2000)	10	+12 Volt
11	Erweiterung	12	GND
13	GND	14	/C3
15	CDAC	16	/C1
17	/OVR	18	XRDY
19	/INT2	20	/PALOPE (unbenutzt b. A2000)
21	A5	22	/INT6
23	A6	24	A4
25	GND	26	A3
27	A2	28	A7
29	A1	30	A8
31	FC0	32	A9
33	FC1	34	A10
35	FC2	36	A11
37	GND	38	A12
39	A13	40	/IPL0
41	A14	42	/IPL1
43	A15	44	/IPL2
45	A16	46	/BERR
47	A17	48	/VPA
49	GND	50	E
51	/VMA	52	A18
53	/RES	54	A19

55	/HLT	56	A20
57	A22	58	A21
59	A23	60	/BR
61	GND	62	/BGACK
63	PD15	64	/BG
65	PD14	66	/DTACK
67	PD13	68	/PRW
69	PD12	70	/LDS
71	PD11	72	/UDS
73	GND	74	/AS
75	PD0	76	PD10
77	PD1	78	PD9
79	PD2	80	PD8
81	PD3	82	PD7
83	PD4	84	PD6
85	GND	86	PD5

Am Expansion-Port liegen so gut wie alle wichtigen Steuerleitungen und Bussignale des Amigasystems an. Dadurch können an ihm RAM-Erweiterungen, neue Prozessoren, Festplatten-Controller usw. angeschlossen werden. Beim A1000 liegt dieser Port neben den beiden Gameports hinter einer leicht abnehmbaren Plastikschutzkappe verborgen. Beim A500 hat man ihn genau gegenüber, d.h. auf der von vorne gesehen linken Gehäusesseite, platziert. Er ist in Form eines 86-poligen Platinensteckers ausgeführt. Der Abstand zwischen den einzelnen Pins beträgt 1/10 Zoll, das sind 2.54 mm. Eine dafür passende Buchse dürfte derzeit nicht leicht zu finden sein.

Beim A2000 gibt es zwei verschiedene Busanschlüsse. Einmal den MMU-Connector, der weitgehend obiger Belegung entspricht (siehe Klammern) und die 5 100poligen Amiga-Steckplätze (auch Zorro-Bus genannt). Diese sechs Steckplätze befinden sich im A2000 auf der Hauptplatine innerhalb des Gehäuses. Sie sind als 86polige bzw. 100polige Buchsen für Platinenstecker ausgeführt. Der Kontaktabstand beträgt ebenfalls 2.54 mm.

Die meisten Signale am Expansion-Port sind direkt mit den entsprechenden Leitungen des 68000 verbunden. Die genaue Funktion dieser Leitungen kann man im Kapitel 1.2.1 nachlesen.

Es sind folgende Signale:

A0-A23: Adreßbus

PD0-PD15: Prozessordatenbus

IPL0-IPL2: Prozessor-Interrupt-Leitungen

FC0-FC2: Function-Code-Leitungen des 68000

AS, UDS, LDS, PRW, DTACK, VMA, VPA: Bussteuer-signale

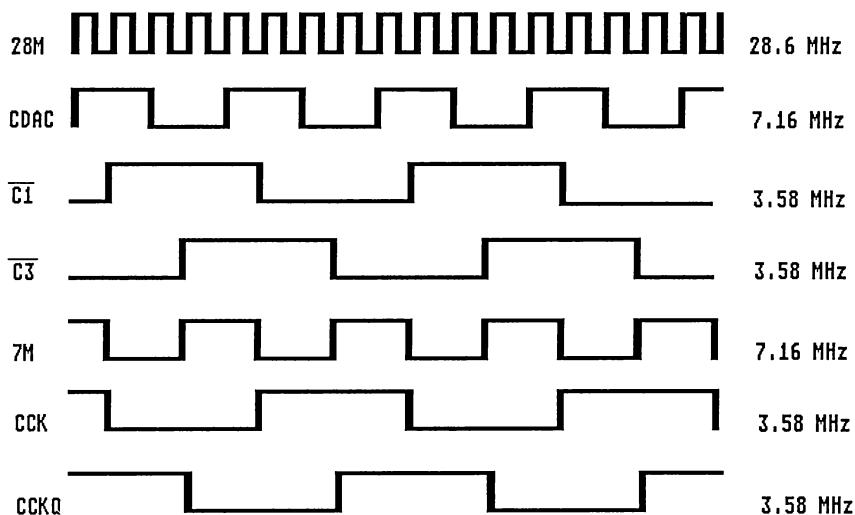
RES, HLT, BERR, BG, BGACK, BR, E: sonstige Steuer-signale des 68000

Die übrigen Signale haben folgende Funktionen:

INT2 und INT6

Diese beiden Leitungen sind mit den gleichnamigen Pins von Paula verbunden. Mit ihnen kann ein Interrupt der Ebene 2 bzw. 6 hervorgerufen werden.

CDAC, C1, C3 und beim A2000 noch 28M



Taktsignale am Expansionport

Abb. 1.3.7.1

Dies sind die verschiedenen Taktsignale des Amiga. Ihre Frequenz und Phasenlage kann man am besten aus der Abbildung entnehmen. Beim A2000 liegt auch noch der Haupttakt des Amiga, die 28.64-MHz-Quarzfrequenz, am Expansion-Port an. Die ebenfalls in der Abbildung eingezeichneten Taktsignale 7M, CCK und CCKQ gibt es nicht am Expansion-Port. 7M ist der Takt des 68000, und CCK und CCKQ sind mit den Customchips verbunden.

XRDY, OVR und PALOPE

Diese Signale dienen der automatischen Konfiguration der angeschlossenen Erweiterungsplatinen. Ihre genaue Funktion konnte bis jetzt leider noch nicht in Erfahrung gebracht werden.

Die mit "Expansion" bezeichneten Leitungen sind nicht belegt. Sie sind künftigen Erweiterungen der Amiga-Hardware vorbehalten. Beim A2000 wurde dies schon teilweise ausgenutzt. Man hat eine der Expansion-Leitungen mit dem 28M-Taktsignal belegt.

1.3.8 Stromversorgung über die Schnittstellen

An sämtlichen Anschlußbuchsen liegen eine oder auch mehrere der drei Betriebsspannungen des Amiga an. Es ist dadurch möglich, Peripheriegeräte über die entsprechende Schnittstelle mit Strom zu versorgen. Allerdings muß man Rücksicht auf die maximale Belastbarkeit der Anschlüsse nehmen. Folgende Tabelle gibt die von Commodore empfohlenen Höchstbelastungen beim A1000 wieder:

Schnittstelle	+5 Volt	+12 Volt	-5 Volt
TV-Mod-Buchse	-	60 mA	-
RGB-Buchse	300 mA	175 mA	50 mA
RS232-Buchse	100 mA	50 mA	50 mA
Ext. Disk	270 mA	160 mA	-
Centronics	100 mA	-	-
Expansion-Port	1000 mA	50 mA	50 mA
Gameport 0	125 mA	-	-
Gameport 1	125 mA	-	-

Diese Tabelle kann allerdings nur als grobe Richtlinie angesehen werden. Erstens gelten die angegebenen Werte nur dann, wenn auch wirklich alle Ports gleichzeitig entsprechend belastet werden. Ist z.B. der Expansion-Port unbelegt, stehen die zusätzlichen 1000 mA an den anderen +5 Volt Anschlüssen zur Verfügung. Ist in einer bestimmten Systemkonfiguration sicher, daß einige Ports frei bleiben, erhöht sich dementsprechend die Belastbarkeit der anderen Ausgänge. Selbstverständlich kann man auch die Gewaltmethode anwenden und solange Expansion-Port-Platinen anschließen, bis das Netzteil abschaltet. Wie (meist unfreiwillige) Kurzschlußversuche gezeigt haben, schadet ihm dies nicht weiter. Allerdings muß jeder selbst die Verantwortung für solche Experimente übernehmen. Vor allem bei den +5 Volt ist

Vorsicht geboten. Bei einem Kurzschluß können Ströme von mehr als 8 Ampere fließen.

Zweitens gilt die obige Tabelle nur für den A1000. Man kann sie nicht auf den A500 oder A2000 übertragen, da die Netzteile dieser Computer unterschiedlich dimensioniert sind. Beim A500 ist die Belastbarkeit des Netzteils geringer. Man sollte bei stromfressenden Erweiterungen am Expansion-Port (RAM-Karten etc.) ein Zusatznetzteil benutzen.

Die Stromversorgung des A2000 ist deutlich kräftiger als die des A1000. Sie muß ja auch in der Lage sein, sämtliche zusätzlichen Amigas und IBM-Steckkarten zu versorgen.

Hinweis: Noch ein entscheidender Unterschied des A500 Netzteils gegenüber dem des A1000 ist die negative Versorgungsspannung. Sie beträgt nämlich -12 statt -5 Volt wie beim A1000. D.h. überall, wo in diesem Buch -5 Volt angegeben sind, müssen A500 Besitzer mit -12 Volt rechnen.

1.4 Die Tastatur

ESC	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	DEL
45	50	51	52	53	54	55	56	57	58	59	46
1 00	2 01	3 02	4 03	5 04	6 05	7 06	8 07	9 08	0 09	= 0A	BACK SPACE 41
TAB	Q	W	E	R	T	Y	U	I	O	P	HELP
42	10	11	12	13	14	15	16	17	18	19	44
CTRL	CAPS LOCK	A	S	D	F	G	H	J	K	L	RETURN
63	62	20	21	22	23	24	25	26	27	28	29
SHIFT	Z	X	C	V	B	N	M	;	'	2	3A
60	30	31	32	33	34	35	36	37	38	39	61
ALT	64	66	40					67	65	ALT	4D

7	0	9
3D	3E	3F
4	5	6
2D	2E	2F
1	2	3
1D	1E	1F
0	.	.
0F	3C	
-	ENTER	
4A	43	

Amerikanische ASCII-Tastatur

ESC	F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	DEL	RETURN
45	50	51	52	53	54	55	56	57	58	59	46	
1 00	2 01	3 02	4 03	5 04	6 05	7 06	8 07	9 08	0 09	= 0A	BACK SPACE 41	
TAB	Q	W	E	R	T	Z	U	I	O	P	44	
42	10	11	12	13	14	15	16	17	18	19	44	5F
CTRL 63	CAPS LOCK 62	A	S	D	F	G	H	J	K	L	4C	
SHIFT 60	Z	X	C	V	B	N	M	;	'	=	SHIFT 61	4C
ALT 64	66	40					67	65	ALT 65	4D		

7	0	9
3D	3E	3F
4	5	6
2D	2E	2F
1	2	3
1D	1E	1F
0	.	.
0F	3C	
-	ENTER	
4A	43	

Deutsche Tastatur

Abb. 1.4.1

Die Amiga-Tastatur ist eine sogenannte intelligente Tastatur. Sie besitzt einen eigenen Mikroprozessor, der die zeitaufwendige Abfrage der einzelnen Tasten übernimmt und dem Amiga die fertigen Tastencodes liefert. Es gibt inzwischen verschiedene Ausführungen und Belegungen der Amiga-Tastatur, aber sie unterscheiden sich nur durch einige neu hinzugekommene Tasten und damit auch Tastencodes. Die Abbildung zeigt die Belegung und die zugehörigen Tastencodes sowohl der deutschen als auch der amerikanischen ASCII-Version. Wie man sieht, entsprechen die Codes **nicht** dem ASCII-Standard. Die Tastatur liefert lediglich sogenannte RAW-Keycodes ("Rohe Tastaturcodes"), die

dann vom Amiga Betriebssystem mit Hilfe einer Übersetzungstabelle, der Keymap, in ASCII-Codes verwandelt werden.

Trotzdem gibt es ein System in der Verteilung der RAW-Keycodes:

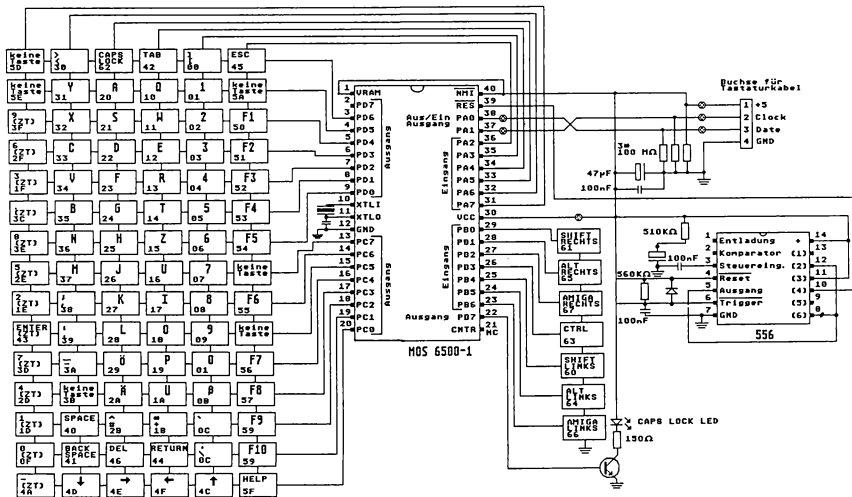
- \$00-\$3F** Dies sind die Codes der Buchstaben-, Ziffern- und Sonderzeichentasten. Ihre Verteilung entspricht der Anordnung der Tasten auf dem Keyboard.
- \$40-\$4F** Die Codes der üblichen Sondertasten wie SPACE, RETURN, TAB usw.
- \$50-\$5F** Die Funktionstasten und HELP
- \$60-\$67** Tasten zur Wahl der unterschiedlichen Tastaturebenen (Shift, Amiga, Alternate und Control)

Der Tastaturprozessor kann allerdings noch mehr. Er unterscheidet nämlich zwischen dem Drücken und Loslassen einer Taste. Wie man sieht, sind sämtliche Tastaturcodes lediglich 7 Bit breit (Wertebereich von \$00 bis \$7F). Das achte Bit ist das KEYup/down-Flag. Es wird von der Tastatur dazu verwendet, um dem Computer anzuzeigen, ob die Taste gerade gedrückt oder losgelassen wurde. Ist das achte Bit 0, bedeutet dies, daß die Taste gerade gedrückt wurde (KEYdown). Ist es auf 1, wurde sie losgelassen (KEYup). Dadurch weiß der Amiga stets, welche Tasten gerade gedrückt sind. Man kann die Tastatur damit auch für Zwecke benutzen, die das gleichzeitige Niederhalten verschiedener Tasten erfordern. Dies sind z.B. Musikprogramme, die die Tastatur zum polyphonen Spielen verwenden wollen.

Eine Ausnahme bildet die CAPS-LOCK-Taste. Die Tastatur simuliert bei ihr einen einrastenden Schalter. Drückt man sie das erste Mal, rastet sie ein, und die Leuchtdiode geht an. Erst wenn man sie ein zweites Mal betätigt, rastet sie wieder aus, und die Leuchtdiode verlöscht. Diesem Verhalten trägt das KEYup/down-Flag Rechnung. Betätigt man CAPS-LOCK, leuchtet die LED auf, und der Tastencode für CAPS-LOCK

wird zusammen mit einem gelöschten 8. Bit zum Computer geschickt, um anzuzeigen, daß die Taste jetzt gedrückt ist. Läßt man die Taste wieder los, wird kein KEYUp-Code gesendet, und die LED leuchtet weiter. Erst wenn man CAPS-LOCK ein weiteres Mal niederdrückt, wird ein KEYUp-Code ausgegeben (gesetztes achttes Bit), und die LED verlöscht.

1.4.1 Die Schaltung der Tastaturplatine



Die Tastenbelegungen beziehen sich auf die DIN-Tastatur
(ZT) bedeutet Zehnertastatur
Alle Tastencodes sind hexadezimal angegeben

Abb. 1.4.2

Kern der Tastaturschaltung ist der Mikroprozessor 6500/1. Der 6500/1 ist ein sogenannter Einchip-Mikrocomputer. Er enthält auf einem Chip alle Komponenten, die ein einfaches Computersystem zum Arbeiten benötigt. Das Herz des 6500/1 ist ein Mikroprozessor vom Typ 6502 (Aha!). Dazu kommen noch 2 KB ROM mit dem Steuerprogramm, 64 Bytes statisches RAM, 4 bi-direktionale 8-Bit-Ports, ein 16-Bit-Zähler mit eigenem Steuereingang und ein Taktgenerator.

Zum Betrieb benötigt der 6500/1 nur noch seine Betriebsspannung von 5 Volt und ein Quarz für die Takterzeugung. In der

Amiga-Tastatur wird der 6500/1 mit einem 3-MHz-Quartz betrieben. Da diese Frequenz intern noch durch zwei geteilt wird, beträgt die Taktfrequenz 1.5 MHz.

Das zweite Chip auf der Tastaturplatine ist ein sogenannter Präzisionszeitgeber vom Typ 556. Genaugenommen befinden sich zwei dieser Präzisionszeitgeber in dem Baustein. Der 556 erzeugt zusammen mit den paar Bauteilen um ihn herum das Reset-Signal für den 6500/1.

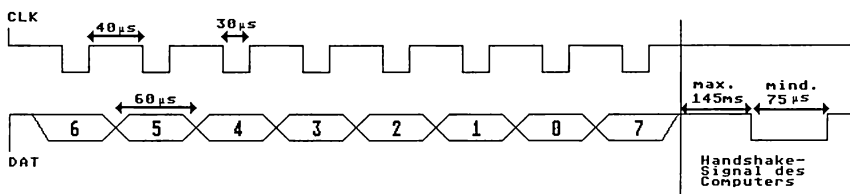
Die Tasten sind in zwei Gruppen aufgeteilt. Die sieben Sonderfunktionstasten (Shift rechts, ALT rechts, Amiga rechts, Control, Amiga links, ALT links und Shift links) sind direkt mit den ersten sieben Portleitungen des PB-Ports verbunden.

Alle restlichen Tasten sind in einer Matrix von sechs Zeilen auf 15 Spalten angeordnet. Die Zeilen sind mit Leitungen PA2 bis PA7 von Port A verbunden. Diese sechs Portleitungen sind als Eingänge geschaltet. Die 15 Spalten werden von Port C und D angesteuert. Die zu PD7 gehörende 16. Spalte ist in den derzeitigen Tastaturversionen nicht beschaltet.

Wenn der 6500/1 die Tastatur abfragt, legt er der Reihe nach die einzelnen Spalten auf 0. Da die Ausgänge von Port C und D OpenCollector-Ausgänge ohne integrierte Pullup-Widerstände sind, verhalten sich alle auf 1 gesetzten Ausgänge vollkommen inaktiv. Nachdem der Prozessor also eine Spalte auf 0 gelegt hat, fragt er die 6 Zeilen ab. Die sechs Zeilen-Eingänge sind mit internen Pullup-Widerständen versehen, so daß alle nicht gedrückten Tasten als High interpretiert werden. Jede gedrückte Taste verbindet immer eine Zeile mit einer Spalte. Sind in der vom 6500/1 gerade aktivierten Spalte Tasten gedrückt, werden die entsprechenden Zeileneingänge auf 0 gelegt. Nachdem alle Spalten einmal aktiviert und die zugehörigen Zeilen gelesen wurden, kennt der Prozessor den Zustand sämtlicher Tasten.

Hat sich dieser seit der letzten Abfrage verändert, sendet er die entsprechenden Tastencodes an den Computer.

1.4.2 Die Datenübertragung



Ausgabe des Datenbytes von Tastaturprozessor

Abb. 1.4.3

Die Tastatur ist mit dem Amiga über ein vieradriges Spiralkabel verbunden. Zwei der Leitungen dienen dabei lediglich der Stromversorgung der Tastaturelektronik mit den nötigen 5 Volt. Die gesamte Datenübertragung läuft über die restlichen beiden Leitungen. Dabei dient die eine als Datenleitung KDAT und die andere als Taktleitung KCLK. Innerhalb des Amiga ist KDAT mit dem seriellen Eingang SP und KCLK mit dem CNT-Pin von CIA-A verbunden (siehe Kap. 1.2.2).

Die Datenübertragung ist unidirektional. Sie läuft immer von der Tastatur zum Amiga. Der 6500/1 legt dazu die einzelnen Datenbits auf die Datenleitung (KDAT), begleitet von 20 Microsekunden langen Low-Impulsen der Clock-Leitung (KCLK). Zwischen den einzelnen Clock-Impulsen liegen jeweils 40 Microsekunden lange Pausen. Das bedeutet, die Übertragungszeit für jedes Bit beträgt 60 Mikrosekunden (20 + 40). Dies ergibt bei 8 Bit 480 Mikrosekunden pro Byte oder umgerechnet eine Übertragungsgeschwindigkeit von 16666 Baud (Bits/Sekunde).

Nach der Ausgabe des letzten Bits erwartet die Tastatur einen Handshake-Impuls vom Computer. Der Amiga legt zu diesem Zweck die KDAT-Leitung für mindestens 75 Mikrosekunden auf Low.

Den genauen Verlauf kann man gut der Abbildung entnehmen.

Allerdings werden die Daten nicht in der üblichen Reihenfolge 7-6-5-4-3-2-1-0 gesendet, sondern vorher noch um eine Bitposition nach links rotiert: 6-5-4-3-2-1-0-7. Z.B. wird aus dem Tastencode für "J" mit gesetztem achten Bit = 10100110 nach dem Rotieren 01001101. Das KEYup/down-Flag wird dadurch immer zuletzt übermittelt.

Zusätzlich ist die Datenleitung noch Low-aktiv. D.h. eine 0 wird durch ein High und eine 1 durch ein Low dargestellt.

Das Schieberegister im CIA des Amiga übernimmt mit jedem Clock-Impuls das aktuelle Bit an der SP-Leitung. Nach acht Clock-Impulsen hat das CIA ein komplettes Datenbyte empfangen. Das CIA erzeugt dann normalerweise einen Interrupt der Ebene 2, der das Betriebssystem dazu veranlaßt, folgende Schritte zu unternehmen:

- Lesen des seriellen Datenregisters des CIA
- Invertieren und Rechtsrotieren des Bytes, um wieder den ursprünglichen Tastencode zu erhalten
- Ausgabe des Handshake-Impulses
- interne Weiterverarbeitung des empfangenen Codes

Synchronisation

Um eine fehlerfreie Datenübertragung durchführen zu können, muß das Timing von Sender und Empfänger übereinstimmen. Die Bitposition bei der seriellen Übertragung muß bei beiden identisch sein. Sonst kann es passieren, daß das Keyboard alle acht Bits ausgegeben hat, während sich der serielle Port des CIAs noch irgendwo mitten im Byte befindet. Ein solcher Verlust der Synchronisation tritt immer dann auf, wenn man den Amiga einschaltet oder die Tastatur in einen bereits laufenden Amiga einsteckt. Der Computer hat keine Möglichkeit, eine fehlerhafte Synchronisation zu erkennen. Diese Aufgabe wird von der Tastatur übernommen.

Nach jedem ausgegebenen Byte wartet die Tastatur maximal 145 Millisekunden auf das Handshake-Signal. Trifft es nicht innerhalb dieses Zeitraums ein, nimmt der Tastaturprozessor an, daß ein Übertragungsfehler vorliegt, und leitet einen speziellen Modus ein, mit dem er versucht, die verlorene Synchronisation wiederzugewinnen. Dazu gibt er immer eine 1 auf der KDAT-Leitung zusammen mit einem Clock-Impuls aus und wartet dann wieder 145 ms auf das Synchronisationssignal. Dies wiederholt er solange, bis er ein Handshake-Signal vom Amiga empfängt. Damit ist die Synchronisation wieder hergestellt.

Allerdings ist das vom Amiga empfangene Datenbyte fehlerhaft. Der Zustand der ersten sieben Bits ist ungewiß. Lediglich das zuletzt empfangene Bit ist sicher eine 1, da der Tastaturprozessor während des oben beschriebenen Vorgangs ja nur Einsen ausgibt. Da dieses letzte Bit das KEYup/down-Flag ist, ist der fehlerhafte Tastencode immer ein KEYup-Code, also eine losgelassene Taste. Dadurch bleiben die möglichen Programmstörungen geringer, als wenn es sich bei dem fehlerhaften Code um einen KEYdown-Code gehandelt hätte. Aus diesem Grund wird jedes Byte vor der Übertragung um ein Bit linksrotiert, damit das KEYup/down-Flag immer das zuletzt gesendete Bit ist.

Die Sondercodes

Es gibt noch einige Sonderfälle in der Übertragung, die die Tastatur dem Amiga mittels spezieller Tastencodes mitteilt. Folgende Tabelle enthält alle möglichen Sondercodes:

Code	Bedeutung
\$F9	Letzter Tastencode war fehlerhaft
\$FA	Tastenpuffer im Keyboard voll
\$FC	Selbsttest der Tastatur war fehlerhaft
\$FD	Beginn der beim Einschalten gedrückten Tasten
\$FE	Ende der beim Einschalten gedrückten Tasten

\$F9

Der Code \$F9 wird von der Tastatur immer nach einem Verlust der Synchronisation und der darauf erfolgten Resynchronisation gesendet. Dadurch erfährt der Amiga, daß der letzte Tastencode fehlerhaft war. Nach diesem Code überträgt die Tastatur den verlorengegangenen Tastencode noch einmal.

\$FA

Die Tastatur verfügt über einen internen Tastenpuffer von 10 Zeichen. Wenn dieser Puffer voll ist, sendet sie eine \$FA zum Computer, um ihm zu signalisieren, daß er den Puffer leeren muß, wenn keine Tastencodes verlorengehen sollen.

\$FC

Nach dem Einschalten führt der Tastaturprozessor einen Selbsttest aus. Man erkennt dies am kurzzeitigen Leuchten der CAPS-LOCK-LED. Entdeckt er dabei einen Fehler, sendet er ein \$FC an den Amiga und geht dann in eine Endlosschleife, in der er die LED blinken läßt.

\$FD & \$FE

War der Selbsttest erfolgreich, überträgt die Tastatur alle Tasten, die während des Einschaltens gedrückt waren. Um dem Computer dies mitzuteilen, beginnt er diese Übertragung mit dem Code \$FD. Es folgen die Codes der während des Einschaltens gedrückten Tasten, und zum Abschluß wird noch ein \$FE zum Amiga geschickt. Danach beginnt die normale Übertragung.

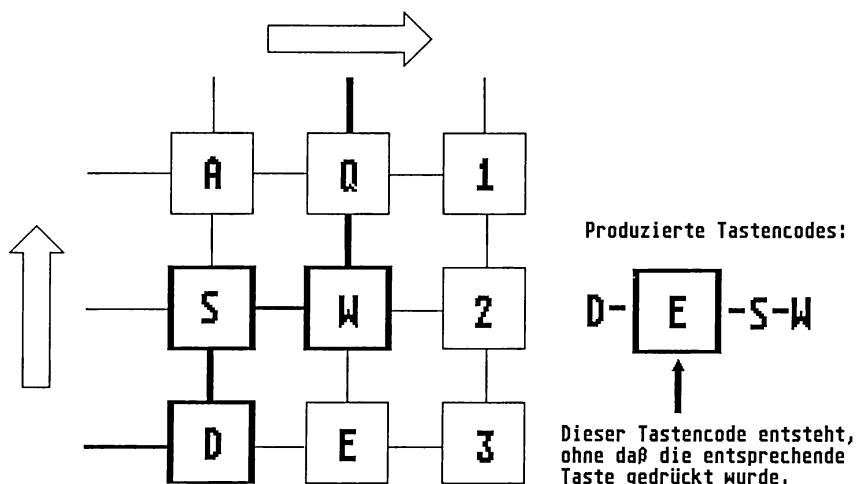
Waren keine Tasten gedrückt, werden \$FD und \$FE unmittelbar hintereinander gesendet.

Reset über die Tastatur

Die Tastatur hat auch noch die Möglichkeit, einen Reset des Amiga auszulösen. Drückt man beide Amiga-Tasten zusammen mit Control, legt der Tastaturprozessor die KCLK-Leitung für etwa 0.5 Sekunden auf Low. Dies veranlaßt die Reset-Schaltung im Amiga dazu, einen Prozessorreset auszulösen. Nachdem man mindestens eine der drei Tasten wieder losgelassen hat, führt die Tastatur auch selbst einen Reset aus. Man erkennt dies am Aufleuchten der CAPS-LOCK-LED. (Interessant ist, daß dieser Reset über die KCLK-Leitung ausgelöst wird, die ja intern mit der CNT-Leitung des CIA-A verbunden ist. Man kann also durch entsprechende Programmierung dieses CIAs softwaremäßig einen Hardware-Reset auslösen.)

1.4.3 Schwächen der Tastatur

Zum Abschluß noch eine Bemerkung zu den Schwächen des Keyboards. Bevor Sie weiterlesen, ein kleines Experiment: Drücken Sie die drei Tasten "A", "Q" und TAB gemeinsam. Keine Angst, der Amiga ist nicht kaputt. Aber es ist doch erstaunlich, daß die CAPS-LOCK-LED aufleuchtet, ohne daß diese Taste gedrückt wurde. Dasselbe funktioniert mit allen anderen Tasten. Wenn man z.B. "S", "W" und "D" zusammen niederhält, wird mit hundertprozentiger Sicherheit auch noch ein "E" auf dem Bildschirm erscheinen.



Die Entstehung zusätzlicher Tastencodes

Abb. 1.4.4

Die Abbildung versucht diesen Sachverhalt verständlich zu machen. Jede gedrückte Taste stellt einen Kurzschluß zwischen einer Spalte und einer Zeile dar. Der Tastaturprozessor steuert eine Spalte an und fragt dann die einzelnen Zeilen ab. Die Pfeile zeigen die Richtung dieser Abfrage. Sie bestimmt die Reihenfolge, in der die gedrückten Tasten erkannt werden, wenn man sie simultan niederdrückt. Wenn jetzt die zum "E" gehörige Spalte angesteuert wird und der Prozessor die entsprechende Zeile abfragt, während "D", "W" und "S" betätigt sind, erkennt er einen Kurzschluß zwischen Zeile und Spalte, den er natürlich für eine gedrückte "E"-Taste hält. Tatsächlich wird dieser Kurzschluß aber von den drei anderen Tasten hervorgerufen, doch der Tastaturprozessor hat keine Möglichkeit, dies zu erkennen. Noch schöner wird es, wenn man fünf Tasten simultan betätigt. Bei der Kombination D, S, A, Q, 1 entstehen vier zusätzliche Codes, nämlich W, 2, E, 3.

Dieser Effekt tritt bei fast allen billigen Matrix Tastaturen auf, er ist also nicht nur auf den Amiga beschränkt.

Man sollte deshalb keine Programme entwerfen, die solche Tastenkombinationen benötigen. Die Amiga-, Shift-, Alternate- und Control-Tasten, die meistens gemeinsam mit anderen Tasten betätigt werden, sind aus diesem Grund auch außerhalb der normalen Tastenmatrix angeordnet.

1.5 Die Programmierung der Hardware

In den vorangehenden Kapiteln wurde der Hardware-Aufbau des Amiga näher betrachtet. Auf den folgenden Seiten geht es nun um die Programmierung der drei Customchips. Nachdem die Hardwareseite klar ist, erfolgt jetzt eine Einführung in die Software, vor allem in die Erzeugung von Grafiken und Tönen.

Die Voraussetzung für eine erfolgreiche Programmierung des Amiga auf Maschinenebene ist die Kenntnis der Speicherbelegung und der Adressen der einzelnen Chip-Register.

1.5.1 Die Speicherbelegung

Die Speicherkonfiguration

Normale Konfiguration		
\$000000	512 KB Chip-RAM	Spiegelung des jeweiligen Bereichs von \$FC0000 - \$FFFFFF
\$080000	Spiegelung des Chip-RAMs	
\$100000	Spiegelung des Chip-RAMs	
\$180000	Spiegelung des Chip-RAMs	
\$200000	8 MB Fast-RAM- Bereich	
\$A00000	CIAs	Basisadresse von CIA-B
		Basisadresse von CIA-A
\$C00000	A500 & A2000 512 KB Erweiterungsram	
\$C80000	Leer	
\$DC0000	A500 & A2000 Echtzeituhr	Basisadresse der Echtzeituhr
\$DF0000	Customchips	Basisadresse der Customchips
\$E00000	Leer	
\$E80000	Bereich der Expansionslots	
\$F00000	ROM-Module	
\$F80000	256 KB Spiegelung des Kickstart-ROMs	
\$FC0000	256 KB Kickstart-ROMs	

Abb 1.5.1

Die erste Grafik zeigt die normale Speicherkonfiguration des Amiga, wie sie sich nach dem Booten dem Programmierer präsentiert. Der gesamte Adreßbereich des 68000 umfaßt 16 Megabyte (Adressen von 0 bis \$FFFFFF). Auf Grund seiner Größe ist es kein Wunder, daß weite Bereiche unbelegt sind oder

manche Chips mehrfach an verschiedenen Adressen auftauchen. Es gab ja keinen Grund, mit Speicher zu geizen. Die Zeiten des vielfachen Bankswitchings sind mit dem 68000 glücklicherweise vorbei.

Das RAM

Der mit Chip-RAM bezeichnete RAM-Bereich enthält den normalen Speicher des Amiga. Ist bei einem A1000 die Speichererweiterung nicht eingesteckt, reicht er nur bis \$3FFFF. Diese 512 KB heißen Chip-RAM, da die drei Customchips nur auf diesen Bereich zugreifen können.

Es ist möglich, daß der Prozessor bei Speicherzugriffen auf das Chip-RAM durch die Aktivitäten der Customchips gebremst wird. Will man dies verhindern, muß man seinen Amiga mit sogenanntem Fast RAM erweitern. Dieses liegt dann ab \$200000 im Speicher und kann bis zu acht Megabyte einnehmen. Da die Customchips auf diesen Bereich keinen Zugriff haben, kann der 68000 hier mit voller Geschwindigkeit arbeiten. Daher leitet sich auch der Begriff Fast RAM ab. Im Grundzustand enthält der Amiga allerdings noch keinerlei Fast RAM.

Die 512-KB-Erweiterungskarte des A500 oder A2000 liegt von \$C00000 bis \$C7FFFF. Sie nimmt eine Sonderstellung ein. Sie ist sozusagen weder Fisch noch Fleisch, d.h. weder richtiges Chip-RAM noch Fast RAM. Einerseits haben die Customchips keinen Zugriff auf diesen Speicherbereich, andererseits wird der Prozessor bei Zugriffen auf diesen RAM-Bereich trotzdem von den Customchips gebremst. Diese RAM-Erweiterung kombiniert also die schlechten Eigenschaften sowohl des Chip-RAM als auch des Fast RAM, ohne deren positive zu übernehmen. Grund dafür ist allerdings nicht eine Bosheit der Amiga-Entwickler, sondern die Einfachheit dieser RAM-Erweiterung und damit deren billige Herstellungsmöglichkeit.

Die CIAs

Die verschiedenen Register der CIAs tauchen innerhalb des Bereichs von \$A00000 bis \$BFFFFFF mehrfach auf. Genauer über die Adressierung der CIAs kann man in Kapitel 1.2 nachlesen. Hier noch einmal die Adressen der einzelnen Register an ihren normalen Positionen:

CIA-A	CIA-B	Name	Funktion
\$BFE001	\$BFD000	PA	Portregister A
\$BFE101	\$BFD100	PB	Portregister B
\$BFE201	\$BFD200	DDRA	Datenrichtungsregister A
\$BFE301	\$BFD300	DDRB	Datenrichtungsregister B
\$BFE401	\$BFD400	TALO	Timer A Lo-Byte
\$BFE501	\$BFD500	TAHI	Timer A Hi-Byte
\$BFE601	\$BFD600	TBLO	Timer B Lo-Byte
\$BFE701	\$BFD700	TBHI	Timer B Hi-Byte
\$BFE801	\$BFD800	E. LSB	Eventcounter Bits 0-7
\$BFE901	\$BFD900	E. MID	Eventcounter Bits 8-15
\$BFEA01	\$BFDA00	E. MSB	Eventcounter Bits 16-24
\$BFEB01	\$BFD800	---	unbenutzt
\$BFEC01	\$BFDC00	SP	serielles Portregister
\$BFED01	\$BFDD00	IRC	Interrupt Kontrollregister
\$BFEE01	\$BFDE00	CRA	Kontrollregister A
\$BFEF01	\$BFDF00	CRB	Kontrollregister B

Die Customchips

Die verschiedenen Register der Customchips nehmen einen Bereich von 512 Bytes ein. Jedes Register hat eine Breite von 2 Bytes (ein Wort). Aus diesem Grund liegen alle Register auf geraden Adressen.

Die Basisadresse des Registerbereichs liegt bei \$DFF000. Die effektive Adresse beträgt demnach \$DFF000 + Registeradresse. Die folgende Liste gibt die Namen und die Funktionen der einzelnen Chip-Register wieder. Es ist klar, daß die meisten Registerbeschreibungen unbekannt sind, da ja über die Funktion der verschiedenen Register noch nichts gesagt worden ist. Aber diese Liste soll auch nur einen Überblick vermitteln und zum Nachschlagen von Registeradressen dienen.

Es gibt vier verschiedene Registertypen:

r (Read)

Dieses Register kann nur gelesen werden.

w (Write)

Dieses Register kann nur beschrieben werden.

s (Strobe)

Ein Zugriff auf ein solches Register löst einen einmaligen Vorgang in dem entsprechenden Chip aus. Dabei ist der Wert des Datenbusses, also das Wort, das in das Register geschrieben werden soll, unerheblich. Diese Register werden gewöhnlich nur von Agnus angesprochen.

er (Early Read)

Ein mit Early Read bezeichnetes Register ist ein DMA-Ausgaberegister. Es enthält die Daten, die per DMA ins Chip-RAM geschrieben werden sollen. Es gibt lediglich zwei solcher Register (DSKDATR und BLTDDAT - Ausgaberegister von Blitter und Disk). Sie werden ausschließlich vom DMA-Controller in Agnus angesprochen, wenn ihr Inhalt ins Chip-RAM geschrieben werden soll. Der Prozessor kann auf diese Register nicht zugreifen.

A, D, P

Diese drei Buchstaben stehen für die drei verschiedenen Chips Agnus, Denise und Paula. Sie geben an, in welchem Chip das entsprechende Register untergebracht ist. Es kann auch möglich sein, daß ein Register in mehreren Chips untergebracht ist. Bei einem Schreibzugriff wird der Wert dann in beiden oder sogar in allen drei Chips gespeichert. Dies ist der Fall, wenn der Inhalt eines bestimmten Registers von mehreren Chips benötigt wird.

Für den Programmierer ist es unwesentlich, in welchem Chip sich ein bestimmtes Register befindet. Er kann den Bereich der Customchips als Ganzes sehen, quasi als ein einziges Chip. Er benötigt nur Adresse und Funktion des gewünschten Registers.

p, d

Ein kleines "d" bedeutet, daß dieses Register nur vom DMA-Controller angesprochen wird. Register mit einem kleinen "p" davor werden nur vom Prozessor oder dem Copper benutzt. Stehen beide Buchstaben bei einem Register, wird dieses zwar gewöhnlich per DMA angesprochen, aber auch ab und zu vom Prozessor.

Registeranzahl: 197

Register, die normalerweise nur vom DMA-Controller angesprochen werden: 54

Basisadresse des Registerbereichs: \$DFF000

Name	Regadr.	Chip	R/W	p/d	Funktion
BLTDDAT	000	A	er	d	Blitter Ausgabedaten (v. Blt. ins RAM)
DMACONR	002	AP	r	p	DMA-Kontrollregister lesen
VPOSR	004	A	r	p	höchstwertiges Bit d. vertikalen Pos.
VHPOSR	006	A	r	p	vertikale und horizontale Strahlpos.
DSKDATR	008	P	er	d	Disk-Lesedaten (von Disk in RAM)
JOY0DAT	00A	D	r	p	Joystick/Mausposition Gameport 0
JOY1DAT	00C	D	r	p	Joystick/Mausposition Gameport 1
CLXDAT	00E	D	r	p	Kollisionsregister
ADKCONR	010	P	r	p	Audio/Disk Kontrollregister lesen
POT0DAT	012	P	r	p	Potentiometer Gameport 0 lesen
POT1DAT	014	P	r	p	Potentiometer Gameport 1 lesen
POTGOR	016	P	r	p	Pot. Port Daten lesen
SERDATR	018	P	r	p	Seriellen Port und Status lesen
DSKBYTR	01A	P	r	p	Diskdatenbyte und Status lesen
INTENAR	01C	P	r	p	Interrupt Enable lesen
INTREQR	01E	P	r	p	Interrupt Request lesen
DSKPTH	020	A	w	p	Disk DMA-Adresse Bits 16-18
DSKPTL	022	A	w	p	Disk DMA-Adresse Bits 1-15
DSKLEN	024	P	w	p	Disk DMA-Blocklänge

DSKDAT	026	P	w	d	Disk Schreibdaten (vom RAM auf Disk)
REFPTR	028	A	w	d	Refresh-Zähler
VPOSW	02A	A	w	p	MSB d. vertikalen Strahlpos. schreiben
VHPOSW	02C	A	w	p	Vert. und horiz. Strahlpos. schreiben
COPCON	02E	A	w	p	Copper Kontrollregister
SERDAT	030	P	w	p	Serielle Daten und Stopbits schreiben
SERPER	032	P	w	p	Ser. Port Kontrollreg. und Baudrate
POTGO	034	P	w	p	Pot. port Daten schreiben und Startbit
JOYTEST	036	D	w	p	In beide Mauszähler schreiben
STREQU	038	D	s	d	horiz. Sync mit VB und Equal Frame
STRVBL	03A	D	s	d	horiz. Sync mit Vertical Blank
STRHOR	03C	DP	s	d	Horizontales Synchronsignal
STRLONG	03E	D	s	d	Kennzeichnung langer horiz. Zeile

Die nun folgenden Register können vom Copper angesprochen werden, wenn COPCON = 1 ist.

BLTCON0	040	A	w	p	Blitterkontrollregister 0
BLTCON1	042	A	w	p	Blitterkontrollregister 1
BLTAFWM	044	A	w	p	Maske für erstes Datenwort von A
BLTALWM	046	A	w	p	Maske für letztes Datenwort von A
BLTCPTH	048	A	w	p	Adresse der Quelldaten C Bits 16-18
BLTCPTL	04A	A	w	p	Adresse der Quelldaten C Bits 1-15
BLTBPTH	04C	A	w	p	Adresse der Quelldaten B Bits 16-18
BLTBPTL	04E	A	w	p	Adresse der Quelldaten B Bits 1-15
BLTAPTH	050	A	w	p	Adresse der Quelldaten A Bits 16-18
BLTAPTL	052	A	w	p	Adresse der Quelldaten A Bits 1-15
BLTDPPTH	054	A	w	p	Adresse der Zieldaten D Bits 16-18
BLTDPTL	056	A	w	p	Adresse der Zieldaten D Bits 1-15
BLTSIZE	058	A	w	p	Startblit und Größe d. Blitterfensters
---	05A				unbenutzt
---	05C				unbenutzt
---	05E				unbenutzt
BLTCMOD	060	A	w	p	Blitter Modulo für Quelldaten C
BLTBMOD	062	A	w	p	Blitter Module für Quelldaten B
BLTAMOD	064	A	w	p	Blitter Modulo für Quelldaten A
BLTDMOD	066	A	w	p	Blitter Module für Zieldaten D
---	068				unbenutzt
---	06A				unbenutzt
---	06C				unbenutzt
---	06E				unbenutzt
BLTCDAT	070	A	w	d	Blitter Quelldatenregister C

BLTBDAT	072	A	w	d	Blitter Quelldatenregister B
BLTADAT	074	A	w	d	Blitter Quelldatenregister A
---	076				unbenutzt
---	078				unbenutzt
---	07A				unbenutzt
---	07C				unbenutzt
DSKSYNC	07E	P	w	p	Disk Synchronisationsmuster

Die nun folgenden Register können in jedem Fall vom Copper beschrieben werden:

COP1LCH	080	A	w	p	Adresse der 1. Copperlist Bits 16-18
COP1LCL	082	A	w	p	Adresse der 1. Copperlist Bits 1-15
COP2LCH	084	A	w	p	Adresse der 2. Copperlist Bits 16-18
COP2LCL	086	A	w	p	Adresse der 2. Copperlist Bits 1-15
COPJMP1	088	A	s	p	Sprung zum Anfang der 1. Copperlist
COPJMP2	08A	A	s	p	Sprung zum Anfang der 2. Copperlist
COPINS	08C	A	w	d	Copper Befehlsregister
DIWSTRT	08E	A	w	p	Obere, linke Ecke des Anzeigefensters
DIWSTOP	090	A	w	p	Untere, rechte Ecke d. Anzeigefensters
DDFSTRT	092	A	w	p	Beginn des Bitplane-DMA's (Horiz. Pos.)
DDFSTOP	094	A	w	p	Ende des Bitplane-DMA's (Horiz. Pos.)
DMACON	096	ADP	w	p	DMA-Kontrollregister schreiben
CLXCON	098	D	w	p	Kollisionskontrollregister schreiben
INTENA	09A	P	w	p	Interrupt enable schreiben
INTREQ	09C	P	w	p	Interrupt request schreiben
ADKCON	09E	P	w	p	Audio, Disk und UART Kontrollregister
AUD0LCH	0A0	A	w	p	Adresse der Audiodaten Bits 16-18
AUD0LCL	0A2	A	w	p	on Tonkanal 0 Bits 1-15
AUD0LEN	0A4	P	w	p	Kanal 0 Länge der Audiodaten
AUD0PER	0A6	P	w	p	Kanal 0 Periodendauer
AUD0VOL	0A8	P	w	p	Kanal 0 Lautstärke
AUD0DAT	0AA	P	w	d	Kanal 0 Audiodaten (zum D/A-Wandler)
---	0AC				unbenutzt
---	0AE				unbenutzt
AUD1LCH	0B0	A	w	p	Adresse der Audiodaten Bits 16-18
AUD1LCL	0B2	A	w	p	von Tonkanal 1 Bits 1-15
AUD1LEN	0B4	P	w	p	Kanal 1 Länge der Audiodaten
AUD1PER	0B6	P	w	p	Kanal 1 Periodendauer
AUD1VOL	0B8	P	w	p	Kanal 1 Lautstärke

AUD1DAT	0BA	P	w	d	Kanal 1 Audiodaten (zum D/A-Wandler)
---	0BC				unbenutzt
---	0BE				unbenutzt
AUD2LCH	0C0	A	w	p	Adresse der Audiodaten Bits 16-18
AUD2LCL	0C2	A	w	p	von Tonkanal 2 Bits 1-15
AUD2LEN	0C4	P	w	p	Kanal 2 Länge der Audiodaten
AUD2PER	0C6	P	w	p	Kanal 2 Periodendauer
AUD2VOL	0C8	P	w	p	Kanal 2 Lautstärke
AUD2DAT	0CA	P	w	d	Kanal 2 Audiodaten (zum D/A-Wandler)
---	0CC				unbenutzt
---	0CE				unbenutzt
AUD3LCH	0D0	A	w	p	Adresse der Audiodaten Bits 16-18
AUD3LCL	0D2	A	w	p	von Tonkanal 3 Bits 1-15
AUD3LEN	0D4	P	w	p	Kanal 3 Länge der Audiodaten
AUD3PER	0D6	P	w	p	Kanal 3 Periodendauer
AUD3VOL	0D8	P	w	p	Kanal 3 Lautstärke
AUD3DAT	0DA	P	w	d	Kanal 3 Audiodaten (zum D/A-Wandler)
---	0DC				unbenutzt
---	0DE				unbenutzt
BPL1PTH	0E0	A	w	p	Adresse von Bitplane 1 Bits 16-18
BPL1PTL	0E2	A	w	p	Adresse von Bitplane 1 Bits 1-15
BPL2PTH	0E4	A	w	p	Adresse von Bitplane 2 Bits 16-18
BPL2PTL	0E6	A	w	p	Adresse von Bitplane 2 Bits 1-15
BPL3PTH	0E8	A	w	p	Adresse von Bitplane 3 Bits 16-18
BPL3PTL	0EA	A	w	p	Adresse von Bitplane 3 Bits 1-15
BPL4PTH	0EC	A	w	p	Adresse von Bitplane 4 Bits 16-18
BPL4PTL	0EE	A	w	p	Adresse von Bitplane 4 Bits 1-15
BPL5PTH	0F0	A	w	p	Adresse von Bitplane 5 Bits 16-18
BPL5PTL	0F2	A	w	p	Adresse von Bitplane 5 Bits 1-15
BPL6PTH	0F4	A	w	p	Adresse von Bitplane 6 Bits 16-18
BPL6PTL	0F6	A	w	p	Adresse von Bitplane 6 Bits 1-15
---	0F8				unbenutzt
---	0FA				unbenutzt
---	0FC				unbenutzt
---	0FE				unbenutzt
BPLCON0	100	AD	w	p	Bitplane Kontrollregister 0
BPLCON1	102	D	w	p	Kontrollregister 1 (Scrollwerte)
BPLCON2	104	D	w	p	Kontrollregister 2 (Prioritätskontr.)
---	106				unbenutzt
BPL1MOD	108	A	w	p	Bitplane Modulo für ungerade Planes
BPL2MOD	10A	A	w	p	Bitplane Modulo für gerade Planes
---	10C				unbenutzt
---	10E				unbenutzt

BPL1DAT	110	D	w	d	Bitplane 1 Daten (z. RGB-Ausgang)
BPL2DAT	112	D	w	d	Bitplane 2 Daten (z. RGB-Ausgang)
BPL3DAT	114	D	w	d	Bitplane 3 Daten (z. RGB-Ausgang)
BPL4DAT	116	D	w	d	Bitplane 4 Daten (z. RGB-Ausgang)
BPL5DAT	118	D	w	d	Bitplane 5 Daten (z. RGB-Ausgang)
BPL6DAT	11A	D	w	d	Bitplane 6 Daten (z. RGB-Ausgang)
---	11C				unbenutzt
---	11E				unbenutzt
SPR0PTH	120	A	w	p	Spritedaten Sprite 0 Bits 16-18
SPR0PTL	122	A	w	p	Spritedaten Sprite 0 Bits 1-15
SPR1PTH	124	A	w	p	Spritedaten Sprite 1 Bits 16-18
SPR1PTL	126	A	w	p	Spritedaten Sprite 1 Bits 1-15
SPR2PTH	128	A	w	p	Spritedaten Sprite 2 Bits 16-18
SPR2PTL	12A	A	w	p	Spritedaten Sprite 2 Bits 1-15
SPR3PTH	12C	A	w	p	Spritedaten Sprite 3 Bits 16-18
SPR3PTL	12E	A	w	p	Spritedaten Sprite 3 Bits 1-15
SPR4PTH	130	A	w	p	Spritedaten Sprite 4 Bits 16-18
SPR4PTL	132	A	w	p	Spritedaten Sprite 4 Bits 1-15
SPR5PTH	134	A	w	p	Spritedaten Sprite 5 Bits 16-18
SPR5PTL	136	A	w	p	Spritedaten Sprite 5 Bits 1-15
SPR6PTH	138	A	w	p	Spritedaten Sprite 6 Bits 16-18
SPR6PTL	13A	A	w	p	Spritedaten Sprite 6 Bits 1-15
SPR7PTH	13C	A	w	p	Spritedaten Sprite 7 Bits 16-18
SPR7PTL	13E	A	w	p	Spritedaten Sprite 7 Bits 1-15
SPR0POS	140	AD	w	dp	Sprite 0 Startposition (vert. und horiz.)
SPR0CTL	142	AD	w	dp	Sprite 0 Kontrollreg. und vert. Stopp
SPR0DATA	144	D	w	dp	Sprite 0 Datenreg. A (z. RGB-Ausg.)
SPR0DATB	146	D	w	dp	Sprite 0 Datenreg. B (z. RGB-Ausg.)
SPR1POS	148	AD	w	dp	Sprite 1 Startposition (vert. und horiz.)
SPR1CTL	14A	AD	w	dp	Sprite 1 Kontrollreg. und vert. Stopp
SPR1DATA	14C	D	w	dp	Sprite 1 Datenreg. A (z. RGB-Ausg.)
SPR1DATB	14E	D	w	dp	Sprite 1 Datenreg. B (z. RGB-Ausg.)
SPR2POS	150	AD	w	dp	Sprite 2 Startposition (vert. und horiz.)
SPR2CTL	152	AD	w	dp	Sprite 2 Kontrollreg. und vert. Stopp
SPR2DATA	154	D	w	dp	Sprite 2 Datenreg. A (z. RGB-Ausg.)
SPR2DATB	156	D	w	dp	Sprite 2 Datenreg. B (z. RGB-Ausg.)
SPR3POS	158	AD	w	dp	Sprite 3 Startposition (vert. und horiz.)
SPR3CTL	15A	AD	w	dp	Sprite 3 Kontrollreg. und vert. Stopp
SPR3DATA	15C	D	w	dp	Sprite 3 Datenreg. A (z. RGB-Ausg.)
SPR3DATB	15E	D	w	dp	Sprite 3 Datenreg. B (z. RGB-Ausg.)
SPR4POS	160	AD	w	dp	Sprite 4 Startposition (vert. und horiz.)

SPR4CTL	162	AD	w	dp	Sprite 4 Kontrollreg. und vert. Stopp
SPR4DATA	164	D	w	dp	Sprite 4 Datenreg. A (z. RGB-Ausg.)
SPR4DATB	166	D	w	dp	Sprite 4 Datenreg. B (z. RGB-Ausg.)
SPR5POS	168	AD	w	dp	Sprite 5 Startposition (vert. und horiz.)
SPR5CTL	16A	AD	w	dp	Sprite 5 Kontrollreg. und vert. Stopp
SPR5DATA	16C	D	w	dp	Sprite 5 Datenreg. A (z. RGB-Ausg.)
SPR5DATB	16E	D	w	dp	Sprite 5 Datenreg. B (z. RGB-Ausg.)
SPR6POS	170	AD	w	dp	Sprite 6 Startposition (vert. und horiz.)
SPR6CTL	172	AD	w	dp	Sprite 6 Kontrollreg. und vert. Stopp
SPR6DATA	174	D	w	dp	Sprite 6 Datenreg. A (z. RGB-Ausg.)
SPR6DATB	176	D	w	dp	Sprite 6 Datenreg. B (z. RGB-Ausg.)
SPR7POS	178	AD	w	dp	Sprite 7 Startposition (vert. und horiz.)
SPR7CTL	17A	AD	w	dp	Sprite 7 Kontrollreg. und vert. Stopp
SPR7DATA	17C	D	w	dp	Sprite 7 Datenreg. A (z. RGB-Ausg.)
SPR7DATB	17E	D	w	dp	Sprite 7 Datenreg. B (z. RGB-Ausg.)
COLOR00	180	D	w	p	Farbpalettenregister 0 (Color table)
COLOR01	182	D	w	p	Farbpalettenregister 1 (Color table)
COLOR02	184	D	w	p	Farbpalettenregister 2 (Color table)
COLOR03	186	D	w	p	Farbpalettenregister 3 (Color table)
COLOR04	188	D	w	p	Farbpalettenregister 4 (Color table)
COLOR05	18A	D	w	p	Farbpalettenregister 5 (Color table)
COLOR06	18C	D	w	p	Farbpalettenregister 6 (Color table)
COLOR07	18E	D	w	p	Farbpalettenregister 7 (Color table)
COLOR08	190	D	w	p	Farbpalettenregister 8 (Color table)
COLOR09	192	D	w	p	Farbpalettenregister 9 (Color table)
COLOR10	194	D	w	p	Farbpalettenregister 10 (Color table)
COLOR11	196	D	w	p	Farbpalettenregister 11 (Color table)
COLOR12	198	D	w	p	Farbpalettenregister 12 (Color table)
COLOR13	19A	D	w	p	Farbpalettenregister 13 (Color table)
COLOR14	19C	D	w	p	Farbpalettenregister 14 (Color table)
COLOR15	19E	D	w	p	Farbpalettenregister 15 (Color table)
COLOR16	1A0	D	w	p	Farbpalettenregister 16 (Color table)
COLOR17	1A2	D	w	p	Farbpalettenregister 17 (Color table)
COLOR18	1A4	D	w	p	Farbpalettenregister 18 (Color table)
COLOR19	1A6	D	w	p	Farbpalettenregister 19 (Color table)
COLOR20	1A8	D	w	p	Farbpalettenregister 20 (Color table)
COLOR21	1AA	D	w	p	Farbpalettenregister 21 (Color table)
COLOR22	1AC	D	w	p	Farbpalettenregister 22 (Color table)
COLOR23	1AE	D	w	p	Farbpalettenregister 23 (Color table)
COLOR24	1B0	D	w	p	Farbpalettenregister 24 (Color table)
COLOR25	1B2	D	w	p	Farbpalettenregister 25 (Color table)
COLOR26	1B4	D	w	p	Farbpalettenregister 26 (Color table)
COLOR27	1B6	D	w	p	Farbpalettenregister 27 (Color table)

COLOR28	1B8	D	w	p	Farbpalettenregister 28 (Color table)
COLOR29	1BA	D	w	p	Farbpalettenregister 29 (Color table)
COLOR30	1BC	D	w	p	Farbpalettenregister 30 (Color table)
COLOR31	1BE	D	w	p	Farbpalettenregister 31 (Color table)

Die Register 1C0 bis 1FC sind unbelegt.

Ein Zugriff auf die Registeradresse 1FE hat keinerlei Funktion zur Folge. Die Chips werden dabei nicht angesprochen (siehe Kapitel 1.2.3).

Das ROM

Die Abbildung 1.5.1 zeigt den ROM-Bereich, wie er nach dem Booten aussieht. Die 256 KB ROM bei \$FC0000 enthalten das Kickstart des Amiga. Der Bereich von \$F80000 bis \$FBFFFF ist mit dem Bereich von \$FC0000 bis \$FFFFFF identisch. In ihm spiegelt sich noch einmal das Kickstart-ROM. Allerdings kann sich diese Konfiguration ändern. Der 68000 holt nach einem Reset die Adresse des ersten Befehls aus der Speicherstelle 4, dem sogenannten Reset-Vektor. Wäre die Speicherkonfiguration unveränderlich, würde der 68000 den Reset-Vektor aus dem Chip-RAM holen, das ja an der Adresse 4 liegt. Da dessen Inhalt nach dem Einschalten unbestimmt ist, würde der Prozessor an eine willkürliche Adresse springen. Die Folge davon wäre, daß das System schon nach dem Einschalten abstürzen würde. Die Lösung sieht folgendermaßen aus: Das Chip, das für die Speicherkonfiguration zuständig ist, hat einen Eingang, der mit der untersten Portleitung von CIA-A (PA0) verbunden ist. Diese sogenannte OVL-Leitung (Memory Overlay) liegt im normalen Betrieb auf 0, und die Speicherkonfiguration entspricht der Abbildung. Nach einem Reset springt die Portleitung automatisch auf 1. Dadurch wird der ROM-Bereich von \$F80000 bis \$FFFFFF in den Adresßbereich von 0 bis \$7FFFF eingblendet. D.h. die Adresse 4 (der Reset-Vektor) entspricht dann der Adresse \$F80004. Dadurch findet der 68000 eine gültige Reset-Adresse, die ihn ins Kickstart springen läßt. Im Laufe der Reset-Routine legt dieses dann die OVL-Leitung auf 0 und schaltet damit auf die normale Speicheranordnung zurück.

Man muß sehr vorsichtig sein, wenn man mit dieser Leitung experimentieren will. Läuft das Programm, das die OVL-Leitung auf 1 setzen soll, im Chip-RAM, kann das katastrophale Auswirkungen haben, denn das Programm schaltet sich quasi selber aus dem Speicher heraus, und der Prozessor landet irgendwo innerhalb des Kickstarts, das ja nach dem Umschalten den Platz des Chip-RAMs einnimmt.

Das A1000-WOM

Weitere Besonderheiten findet man bei den A1000-Modellen. Besitzer dieser Geräte haben sich sicher schon gewundert, daß hier andauernd von einem Kickstart-ROM gesprochen wird, obwohl der Amiga das Kickstart ja am Anfang von Diskette lädt. Nun, die Situation beim A1000 war folgende: Die Hardware war fertig, die Geräte verkaufsbereit, aber mit der Software, in Form des Betriebssystems Kickstart, lag es noch im argen. Es war noch unvollständig und immer noch mit Fehlern behaftet. Also beschloß man, den Amiga mit einem speziellen RAM zu versehen, in welches nach dem Einschalten das Betriebssystem geladen wurde. Danach verriegelte der Amiga den Schreibzugriff auf dieses RAM. Es verhielt sich nun wie ein 256 KB großes ROM. Man gab ihm bei Commodore den Namen WOM (Write Once Memory / nur einmal beschreibbarer Speicher). Jetzt konnte man die ersten Amigas mit der noch unvollständigen Kickstart-Version (1.0) ausliefern. Nach der Fertigstellung neuerer Kickstart-Versionen (1.1 und 1.2) brauchten die Amiga-Benutzer lediglich die neuen Kickstartdisketten einlegen.

Da dieses WOM natürlich teurer ist als ein einfacher ROM-Baustein, rüstete man den A500 und A2000 nicht mehr damit aus, da ja auch inzwischen eine endgültige Kickstartversion, V1.2, fertig war.

Das WOM wirft aber noch einige Fragen auf: Wo ist das Programm, das nach dem Einschalten Kickstart lädt? Wie kann ich Kickstart ändern, wenn es ja in einem RAM liegt?

Im Normalfall sieht der Betriebssystembereich im A1000 genauso wie in den neueren Modellen aus. Kickstart von \$FC0000 bis

\$FFFFFF mit Spiegelung bei \$F80000. Versucht man, in das Kickstart zu schreiben, passiert gar nichts. Es ist kein Schreibzugriff möglich. Auch das sogenannte Boot-ROM, das Kickstart am Anfang lädt, ist nirgendwo eingeblendet.

Gesteuert wird das Ganze nämlich über die Reset-Leitung. Nach einem Reset, egal ob beim Einschalten, durch Drücken von Amiga & Control oder durch einen Reset-Befehl des 68000, ändert sich die Speicherkonfiguration.

Danach liegt das Boot-ROM bei \$F80000 (Da bei einem Reset gleichzeitig die OVL-Leitung gesetzt wird, stammt auch der Reset-Vektor aus dem Boot-ROM), und es ist möglich, ins Kickstart zu schreiben! Man kann es jetzt nach Belieben verändern. Dieser Zustand bleibt aber nur solange erhalten, bis man versucht, etwas in den Bereich des Boot-ROMs von \$F80000 bis \$FBFFFF zu schreiben. Dann wird das Boot-ROM wieder ausgeblendet und der Schreibzugriff auf Kickstart verboten. Kurz gesagt:

Reset erlaubt das Schreiben ins Kickstart und blendet das Boot-ROM ein.

Ein Schreibzugriff auf eine Adresse zwischen \$F80000 und \$FBFFFF verbietet das Schreiben und schaltet das Boot-ROM aus.

1.5.2 Grundlagen

Wie in dem vorangegangenen Kapitel erwähnt wurde, gibt es Register, die vom Prozessor angesprochen werden, und solche, die per DMA gelesen und beschrieben werden. Gehen wir zuerst auf den ersten Fall ein.

Die Programmierung der Chip-Register

Die Chip-Register können direkt adressiert werden. Beispiel: Der Wert des Hintergrundfarben-Registers soll verändert werden. Das Register hat den Namen COLOR00. Wenn man in der Tabelle aus 1.5.1 nachschlägt, findet man eine Registeradresse von \$180. Dazu kommt noch die Basisadresse des Registerbereichs, d.h. die Adresse des ersten Registers auf den Adreßbereich des 68000 bezogen. Sie beträgt \$DFF000. Plus der Registeradresse von COLOR00 ergibt \$DFF180. Ein einfacher MOVE.W-Befehl genügt zur Initialisierung des Registers:

```
MOVE.W #Wert,$DFF180 ;Wert in COLOR00
```

Sollen mehrere Register angesprochen werden, empfiehlt es sich, die Basisadresse in einem Adreßregister unterzubringen und die indirekte Adressierung plus Offset zu verwenden. Dazu ein Beispiel:

```
LEA $DFF000,A5 ;Basisadresse in A5 speichern  
MOVE.W #Wert1,$180(A5) ;Wert1 in COLOR00  
MOVE.W #Wert2,$182(A5) ;Wert2 in COLOR01  
MOVE.W ... usw.
```

Normalerweise findet der Zugriff auf ein Chip-Register wie oben gezeigt statt. Allerdings kann man die Register auch als Langwort ansprechen. In diesem Fall werden dann immer zwei Register auf einmal beschrieben. Dies hat seinen Sinn bei den sogenannten Adreßregistern. Diese Register bestehen aus einem Registerpaar, das eine 19 Bit Adresse enthält, mit der der ganze Chip-RAM-Bereich von 512 KB angesprochen werden kann. Alle mit den Customchips zusammenhängenden Daten müssen im Chip-RAM liegen. Da die Chips den Speicher immer nur wortweise adressieren, ist das unterste Bit (Bit 0) unwesentlich. Das Adreßregister zeigt nur auf gerade Adressen. Da ein Chip-Register nur ein Wort, d.h. 16 Bit, breit ist, dienen zwei Register an aufeinanderfolgenden Registeradressen gemeinsam zur Aufnahme der 19 Bit Speicheradresse. Dabei enthält das erste die oberen 3 Bit (Bits 16 bis 18) und das zweite die unteren 16 (Bits 0 - 15). Dadurch ist es möglich, mit einem einzigen Langwort-Zugriff beide Register zu initialisieren. Beispiel: Es soll der Zei-

ger für die erste Bitplane auf die Adresse \$40000 gesetzt werden. BPL1PTH ist der Name des ersten Registers (Bits 16-18) und BPL1PTL (Bits 0-15) der des zweiten. Registeradresse von BPL1PTH: \$0E0, BPL1PTL = \$0E2.

A5 enthält die Basisadresse \$DFF000.

```
MOVE.L #$40000,$0E0(A5) ;initialisiert BPL1PTH und BPL1PTL mit den  
                           ;korrekten Werten.
```

Es muß nochmals darauf hingewiesen werden, daß man ein Register nicht an ein und derselben Registeradresse lesen und beschreiben kann. Die meisten Register sind sowieso Nur-Schreib-Register. Sie können nicht gelesen werden. Dazu gehören auch die oben verwendeten Register. Andere können ausschließlich gelesen werden. Nur einige wenige können sowohl gelesen als auch beschrieben werden. Sie besitzen dann aber zwei unterschiedliche Registeradressen. Eine zum Lesen und eine zum Schreiben. Das DMA-Kontrollregister, das nachher noch näher besprochen werden soll, ist beispielsweise ein solches Register. Beschreiben kann man es an der Registeradresse \$096 (DMACON). Zum Lesen muß man die Adresse \$002 verwenden (DMACONR - Der Zusatz R steht für read / lesen).

Der DMA-Zugriff

Unter DMA versteht man, wie schon in Kapitel 1.2.3 beschrieben, den direkten Zugriff eines Peripheriechips, des sogenannten DMA-Controllers, auf den Systemspeicher. Dies besagt auch die englische Bezeichnung: DMA - Direct Memory Access / Direkter Speicherzugriff. Im Falle des Amiga, ist der DMA-Controller innerhalb von Agnus untergebracht. Er stellt die Verbindung der unterschiedlichen Ein-/Ausgabe-Einheiten (engl. Input/Output, kurz I/O) der Customchips mit dem Chip-RAM dar. Egal ob es um Disketten, Bildschirm oder Audiodaten geht, der DMA-Vorgang läuft immer nach dem gleichen Muster ab. Eine beliebige Ein-/Ausgabe-Einheit, z.B. der Diskcontroller, benötigt neue Daten oder hat Daten bereit, die in den Speicher müssen. Der DMA-Controller wartet dann auf den richtigen Zeitpunkt, an dem der Speicher für diesen DMA-Kanal frei, d.h. nicht von

einem anderen DMA-Kanal oder dem Prozessor belegt ist, und überträgt die Daten selbständig ins RAM oder liest sie aus. Der Einfachheit halber gibt es keine spezielle Übertragung der Daten von der Ein-/Ausgabeeinheit zum DMA-Controller. Sie läuft ganz normal über Register ab. Jede dieser I/O-Einheiten besitzt zwei unterschiedliche Registertypen. Einmal die normalen Register, die vom Prozessor angesprochen und in denen die verschiedenen Betriebsparameter gespeichert werden, und andererseits die Datenregister, die die Daten für den DMA-Controller enthalten. Dieser spricht bei einem DMA-Transfer einfach simultan das entsprechende Datenregister und die zugehörige RAM-Speicherstelle an. Je nach der Richtung des DMA-Transfers entweder ein Leseregister und Chip-RAM auf Schreiben oder ein Schreibregister und Chip-RAM auf Lesen. Da beide dann über den Datenbus miteinander verbunden sind, wandern die Daten automatisch vom Datenregister ins RAM oder umgekehrt. Die Daten werden nicht in irgendwelchen internen Registern zwischengespeichert.

Durch den DMA-Transfer kommt noch ein dritter Registertyp dazu: Die DMA-Adreßregister, die, je nach den Bedürfnissen der zugehörigen I/O-Einheit, die Adresse/Adressen der Daten im RAM enthalten.

Über all dem stehen dann noch die zentralen Kontrollregister, die nicht einer speziellen Ein-/Ausgabe-Einheit zugeordnet sind, sondern übergeordnete Steuerfunktionen haben. Unter diese Kategorie fällt auch oben erwähntes DMACON-Register.

Die Datenregister können natürlich auch vom Prozessor beschrieben werden, da sie ja in Form normaler Register realisiert wurden. Dies hat aber für gewöhnlich keinen Sinn, da der DMA-Controller diese Aufgabe schneller und eleganter erledigt.

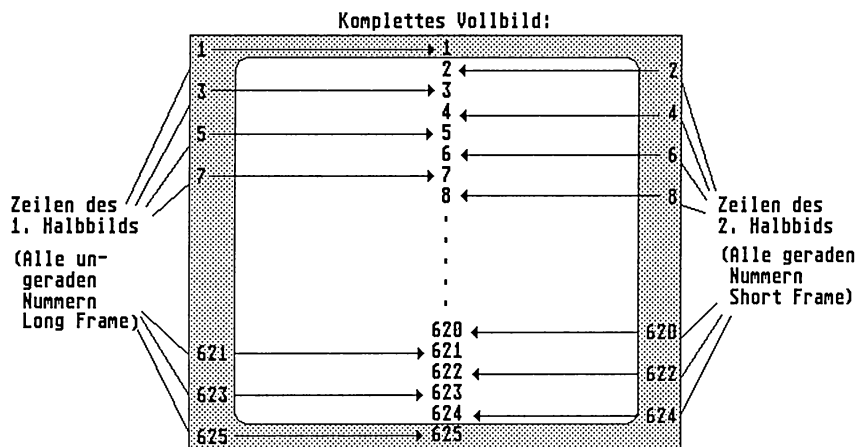
Einige Ein-/Ausgabeeinheiten besitzen keinen zugehörigen DMA-Kanal. Bei ihnen muß der 68000 die Daten selber lesen bzw. schreiben. Dazu gehören aber nur jene Einheiten, bei denen von Natur aus keine hohen Datenmengen anfallen, ein DMA also nicht nötig ist, wie z.B. die Joystick- und Mauseingänge.

Folgende DMA-Kanäle sind vorhanden:

Bitebenen-DMA	Über diesen DMA-Kanal werden die Bildschirmdaten aus dem Speicher gelesen und in die Datenregister der einzelnen Bitebenen geschrieben, von wo aus sie in die Bitebenen-Sequencer gelangen, die die Daten für die Ausgabe auf den Bildschirm umwandeln.
Sprite-DMA	Übertragung der Sprite-Daten aus dem RAM in die Sprite-Datenregister.
Disk-DMA	Daten von der Diskette ins RAM oder vom RAM auf die Diskette.
Audio-DMA	Liest die digitalen Tondaten aus dem RAM und schreibt die in die entsprechenden Audiodatenregister.
Copper-DMA	Über ihn erhält der Coprozessor (Copper) seine Befehlsworte
Blitter-DMA	Daten von und zum Blitter

Es gibt also sechs DMA-Kanäle, die alle auf den Speicher zugreifen wollen, plus dem Prozessor, der natürlich auch möglichst oft das Chip-RAM für sich haben will. Um die dadurch entstehenden Schwierigkeiten zu lösen, hat man ein kompliziertes Zeitmuster entworfen, in dem man den einzelnen Kanälen feste Positionen zugewiesen hat. Da sich dieses an dem Fernsehbild orientiert, müssen wir zuerst kurz auf dessen Aufbau eingehen. Dieser Abschnitt ist so untechnisch wie möglich gehalten, denn es geht in diesem Kapitel ja ums Programmieren der Custom-chips, nicht mehr um die Hardware.

Der Aufbau des Fernsehbilds



Aufbau eines PAL-Fernsehbilds

Abb. 1.5.2.1

Das Zeitverhalten der Bildschirmausgabe eines deutschen Amigas richtet sich exakt nach der deutschen PAL-Fernsehnorm. Ein solches Fernsehbild setzt sich aus 625 horizontalen Zeilen zusammen. Jede dieser Zeilen wird von links nach rechts aufgebaut. Nach jeder Zeile folgt eine Pause, die sogenannte horizontale Austastlücke, in der der Elektronenstrahl, der das Bild zeichnet, Zeit hat, wieder von rechts nach links zurückzulaufen. Während dieser Austastphase ist der Elektronenstrahl dunkel geschaltet, damit der Strahlrücklauf nicht in Form störender Streifen sichtbar wird. Danach beginnt der Vorgang wieder von neuem, und die nächste Zeile entsteht.

Damit das Bild flimmerfrei ist, muß es ständig neu gezeichnet werden. Da unser Auge Bildwechsel oberhalb einer bestimmten Frequenz nicht mehr einzeln wahrnehmen kann, hat man die Anzahl der Bilder pro Sekunde über diese Grenze gelegt. Bei der PAL-Norm liegt die Anzahl der einzelnen Bilder auf 50 pro Sekunde. Allerdings kommt jetzt ein Umstand hinzu, der die ganze

Sache kompliziert. Würde man nämlich sämtliche 625 Zeilen 50 Mal in der Sekunde zeichnen, käme man auf 31250 Zeilen pro Sekunde. Als die Grundlagen dieses Fernsehsystems geschaffen wurden, war man nicht in der Lage, eine solch hohe Zeilenfrequenz mit preisgünstigen, für jeden erschwinglichen Fernsehapparaten zu verarbeiten. Also behelf man sich mit einem Trick. Einerseits sollte die Anzahl der Bilder nicht unter 50 pro Sekunde liegen, da dies ein starkes Flimmern zur Folge hätte, andererseits durfte die Zeilenzahl eines Bildes nicht zu niedrig liegen. Die Lösung sah folgendermaßen aus: Es werden weiterhin 50 Bilder pro Sekunde dargestellt. Allerdings werden die 625 Zeilen auf zwei Bilder verteilt. Im ersten Bild werden alle ungeraden Zeilen übertragen (Zeilen 1, 3, 5 ... 625), im zweiten dann alle geraden (2, 4, 6 ... 624). Folgerichtig gab man den 50 Bildern pro Sekunden den Namen Halbbilder, da sie ja immer nur die Hälfte aller Zeilen enthalten. Zwei Halbbilder zusammen ergeben dann das Vollbild, das jetzt alle 625 Zeilen enthält. Die Anzahl der Vollbilder pro Sekunde ist natürlich nur halb so groß wie die Anzahl der Halbbilder, nämlich 25 pro Sekunde. Die Zeilenfrequenz beträgt durch diese Technik nur noch 15625 Hz ($25 \cdot 625$ oder $50 \cdot 312,5$).

Trotz der hohen Auflösung von 625 Zeilen tritt ein Flimmern nur dann auf, wenn eine Kontur lediglich auf eine Zeile beschränkt ist. Sie wird dann nur noch alle 25stel Sekunde einmal dargestellt, was für das Auge ein deutliches Flimmern zur Folge hat. Diesen Effekt kann man beim Fernsehen besonders an den horizontalen Rändern von Flächen beobachten, da diese naturgemäß aus einer einzigen horizontalen Zeile bestehen.

Die englische Bezeichnung für diese Technik der abwechselnden Übertragung von geraden und ungeraden Zeilen lautet Interlace. Zwei weitere Begriffe dienen der Unterscheidung der beiden Typen von Halbbildern. Mit Long Frame bezeichnet man jenes, in dem die ungeraden Zeilen dargestellt werden, und mit Short Frame das andere. Long Frame und Short Frame deshalb, weil es eine ungerade Zeile mehr als gerade gibt, und dementsprechend das Halbbild mit den ungeraden Zeilen zeitlich gesehen länger ist. (Von 1 bis 625 gibt es 313 ungerade und 312 gerade Zahlen.)

Nach jedem Halbbild folgt eine Pause, bevor das nächste Halbbild beginnt. Diese Schwarzphase zwischen zwei Halbbildern ist die vertikale Austastlücke.

Auch das vom Amiga erzeugte Bild richtet sich nach obigen Grundsätzen. Allerdings mit kleinen Abweichungen.

Normalerweise beginnt das zweite Halbbild (Short Frame) etwas verzögert, so daß die geraden Zeilen genau zwischen die ungeraden zu liegen kommen und sich ein geschlossenes Bild ergibt.

Beim Amiga dagegen sind beide Halbbilder identisch. Dadurch liegt die Bildfrequenz bei tatsächlichen 50 Hz. Als Folge davon ist die Zeilenzahl auf 313 beschränkt. Man erkennt auch deutlich die Zwischenräume zwischen zwei Zeilen auf dem Monitor, da die Halbbilder nicht mehr versetzt zu einander gezeichnet werden.

Um die Zeilenzahl zu erhöhen, hat der Amiga aber auch die Möglichkeit, sein Bild im Interlace-Modus zu erzeugen. Dann sind volle 625 Zeilen möglich. Man muß aber die Nachteile des Interlace-Betriebs mit in Kauf nehmen. Doch dazu später.

Der Aufbau der Amiga-Bildschirmausgabe

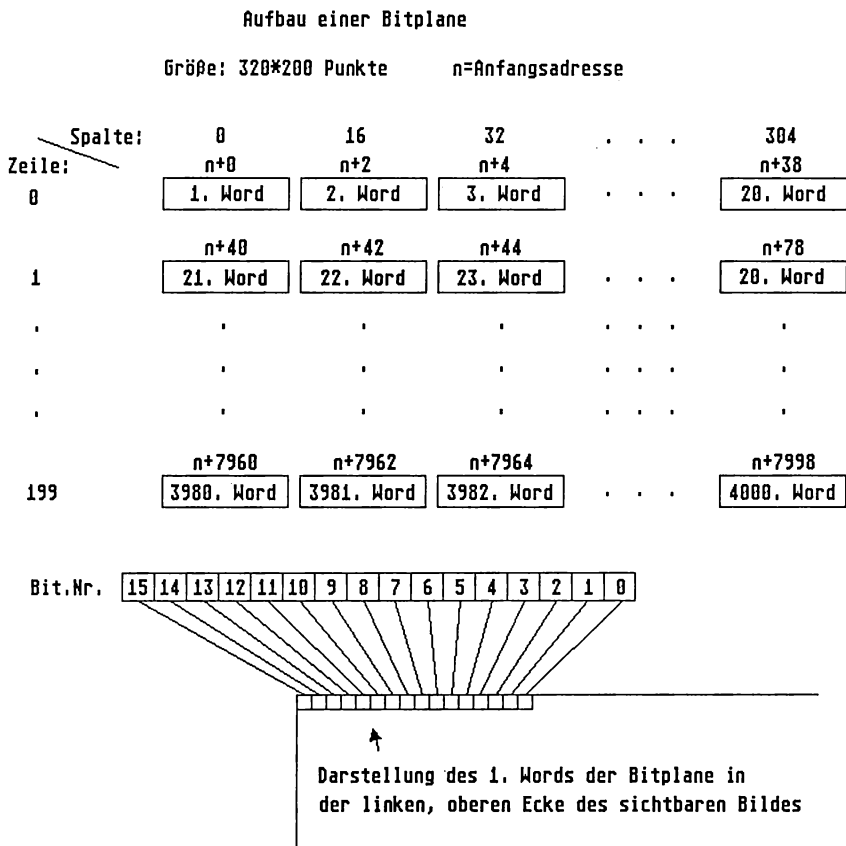


Abb. 1.5.2.2

Bitplanes

Der Amiga stellt sein Bild immer in einer Art Grafikmodus dar, d.h. jeder Punkt auf dem Bildschirm hat seine Entsprechung im Speicher. Im einfachsten Fall entspricht ein gesetztes Bit im RAM einem Punkt auf dem Monitor. Dieser einfachste Aufbau eines Bildschirmspeichers heißt beim Amiga Bitebene oder eng-

lisch Bitplane. Sie ist das Grundelement jeder Bildschirmdarstellung beim Amiga. Sie besteht aus einem zusammenhängenden Speicherbereich. Je nach Bildbreite ergeben eine bestimmte Anzahl von Wörtern eine Bildschirmzeile. Ein Wort entspricht 16 Punkten, da ja jedes Bit einen Bildpunkt repräsentiert. Für eine Bildschirmdarstellung mit 320 Punkten pro Zeile benötigt man also $320/16 = 20$ Wörter. Da man mit einer Bitplane ja nur zwei Zustände unterscheiden kann, nämlich Punkt gesetzt oder Punkt gelöscht, gibt es die Möglichkeit, mehrere Bitplanes zu kombinieren. Dabei werden immer die Bits mit den gleichen Positionen in allen Planes zusammengefaßt. Der erste Punkt auf dem Bildschirm entsteht durch Kombination des obersten Bits im ersten Wort sämtlicher Planes. Der aus diesen Bits resultierende Wert bestimmt dann die Farbe des Punkts auf dem Bildschirm. Um von der Bitkombination eines Punkts zu seiner Farbe auf dem Bildschirm zu kommen, gibt es verschiedene Möglichkeiten, auf die in Kapitel 1.5.5 noch detailliert eingegangen wird.

Die verschiedenen Grafikauflösungen

Der Amiga kennt zwei verschiedene horizontale Auflösungen. Der hochauflösende Modus hat normalerweise 640 Punkte pro Zeile, der niedrigauflösende 320. Das "normalerweise" im letzten Satz bedeutet, daß dieser Wert veränderlich ist. Besser ist es, wenn man die beiden unterschiedlichen Auflösungen durch die Zeit pro Bildpunkt definiert. Ein Pixel (= Bildpunkt) im hochauflösenden Modus wird 70 Nanosekunden (Milliardstel Sekunden) lang angezeigt, im niedrigauflösenden Modus 140 Nanosekunden. In der doppelten Zeit legt der Elektronenstrahl die zweifache Strecke auf dem Bildschirm zurück. Aus diesem Grund sind die niedrigauflösenden Pixel auch doppelt so breit.

Wichtiger für den Programmierer ist es aber zu wissen, daß im hochauflösenden Modus lediglich vier Bitplanes gleichzeitig aktiviert sein dürfen, während im niedrigauflösenden bis zu 6 Planes erlaubt sind.

Der Aufbau einer horizontalen Rasterzeile

Mit dem Begriff Rasterzeile ist eine komplette horizontale Zeile gemeint, d.h. die horizontale Austastlücke und der sichtbare Bereich. Diese Rasterzeile dient als Zeitmaß für alle DMA-Vorgänge, insbesondere natürlich für die mit dem Bildschirm zusammenhängenden DMAs. Um die Aufteilung der Rasterzeile zu verstehen, muß man wissen, wie die Speicherzugriffe auf das Chip-RAM und die CustomChip-Register zwischen dem DMA-Controller und dem Prozessor verteilt werden. Die Zugriffe auf diese beiden Speicherbereiche müssen sich nach den sogenannten Buszyklen richten. Die Buszyklen bestimmen das Timing des Chip-RAMs. In jedem Buszyklus kann ein Speicherzugriff stattfinden. Ob die Daten gelesen oder beschrieben werden, ist dabei unwesentlich. Will z.B. der Prozessor auf den Bus zugreifen, bekommt er den Bus für einen Buszyklus zugewiesen. Der DMA-Controller kann dann erst wieder in dem darauffolgenden Zyklus auf das RAM zugreifen. Ein Buszyklus dauert 280 Nanosekunden. Innerhalb einer Mikrosekunde sind also knapp vier Speicherzugriffe möglich.

Der 68000 kann aber gar nicht so oft auf den Speicher zugreifen. Er ist dazu einfach nicht schnell genug. Bei der Taktfrequenz, mit der er im Amiga betrieben wird, führt er höchstens alle 560 Nanosekunden einen Zugriff aus. In der gleichen Zeit laufen aber zwei Buszyklen ab. Der 68000 kann also maximal jeden zweiten Buszyklus belegen. Diese Zyklen werden gerade Speicherzyklen (Even Cycles) genannt. Die übrigen, ungeraden Zyklen (Odd Cycles) sind damit ausschließlich für den DMA-Controller da.

dieser die geraden mit dem Prozessor teilen. Dabei haben die DMA-Zugriffe immer Vorrang. Der Blitter-DMA und der Copper-DMA finden ausschließlich während gerader Zyklen statt. Allerdings gibt es für die beiden keine bestimmten zeitlichen Festlegungen. Der Copper-DMA belegt alle geraden Speicherzyklen, bis er seine Aufgabe beendet hat. Er hat dabei Vorrang vor dem Blitter. Auch der Blitter belegt alle geraden Zyklen, bis er fertig ist. Es gibt hier allerdings die Möglichkeit, einige Zyklen für den 68000 frei zu halten.

Wie man sieht, belegen Disketten-, Audio- und Sprite-DMA lediglich ungerade Buszyklen, beeinflussen also nicht die Geschwindigkeit des Prozessors. Die vier mit "R" bezeichneten Buszyklen sind die sogenannten Refresh-Zyklen. Sie dienen dazu, den Speicherinhalt des Chip-RAMs aufzufrischen (siehe Ende dieses Kapitels).

Etwas komplizierter ist die Verteilung der Bitplane-DMA's. Um die ersten 16 Punkte auf dem Bildschirm darstellen zu können, müssen alle Bitplanes gelesen werden. Während diese 16 Pixels auf dem Bildschirm erscheinen, müssen schon alle Bitplanes für die nächsten 16 Punkte gelesen werden. Ist die niedrige Auflösung eingeschaltet, werden in jedem Buszyklus 2 Punkte ausgegeben. Dies bedeutet, daß alle acht Buszyklen die Bitplanes gelesen werden müssen. Solange nicht mehr als vier Bitplanes aktiviert sind, reichen dazu die ungeraden Zyklen aus. Werden aber fünf oder sechs Planes benutzt, müssen auch zwei gerade Zyklen verwendet werden, damit alle Daten innerhalb der acht Zyklen gelesen werden können. Noch enger wird es bei einer hochauflösenden Darstellung. Hier werden 4 Punkte pro Buszyklus dargestellt. Sollen nur die ungeraden Zyklen belegt werden, dürfen nicht mehr als zwei Hires-Bitplanes aktiv sein. Bei der maximal möglichen Anzahl von vier Hires-Bitplanes sind alle Buszyklen belegt. Der Prozessor verliert dadurch mehr als die Hälfte seiner freien Buszyklen! Seine Geschwindigkeit nimmt dadurch etwa um den gleichen Faktor ab, vorausgesetzt, daß sich das aktuelle Programm im Chip-RAM befindet, denn auf eventuelles Fast RAM und auf das Kickstart-ROM hat der Prozessor immer noch ungebremsten Zugriff.

Die mit Datafetch-Start und Datafetch-Stop bezeichneten Zeitpunkte stellen den Beginn und das Ende der DMA-Zugriffe für die Bitplanes dar. Sie bestimmen damit die Breite und die horizontale Position des sichtbaren Bildes. Setzt der Bitplane-DMA früh ein und hört spät auf, werden mehr Datenwörter gelesen und damit auch mehr Punkte ausgegeben. Die normale Auflösung von 320 bzw. 640 Punkten pro Zeile läßt sich durch Ändern dieser Werte variieren. Setzt man den Datafetch-Start Wert kleiner \$30, benutzt der Bitplane-DMA-Kanal die normalerweise für den Sprite-DMA reservierten Zyklen. Dadurch fallen je nach Wert von Datafetch-Start bis zu sieben Sprites aus. Lediglich Sprite 0, der ja im allgemeinen als Maus-Zeiger verwendet wird, läßt sich nicht auf diese Weise abschalten.

Die obere Zeile in der Abbildung stellt die Verteilung der DMA-Zyklen bei einem normalen, 320 Punkte breiten, niedrig auflösenden Bildschirm dar. Der Beginn des Bitplane-DMA, Datafetch-Start, liegt bei \$38, und das Ende, Datafetch-Stop, bei \$D0. In den mit "L1" bezeichneten Zyklen werden die Daten der Bitplane Nr. 1 gelesen, in "L2" dann Bitplane 2 usw. Sind die entsprechenden Bitplanes nicht eingeschaltet, fallen auch die zugehörigen DMA-Zyklen weg.

Die zweite Zeile stellt den Ablauf einer Rasterzeile dar, in dem die Datafetch-Punkte nach außen verlegt wurden. Bis zum Datafetch-Start ist der Verlauf mit der oberen Zeile identisch, bei \$28 beginnt dann der Bitplane-DMA. Als Folge davon fallen die Sprites Nr. 5 bis 7 aus. Die Datafetch-Stop-Position wurde bis an den Maximalwert \$D8 nach rechts verschoben.

Die dritte Zeile zeigt die Verteilung der DMA-Zyklen in einem hochauflösenden Bildschirm, wobei die Datafetch-Werte mit denen der ersten Zeile übereinstimmen.

Während der vertikalen Austastlücke finden keine Bitplane-DMA-Zugriffe statt.

Das DMA-Kontrollregister

Die einzelnen DMA-Kanäle werden über ein zentrales DMA-Kontrollregister, DMACON, ein- und ausgeschaltet.

DMACON Registeradr. \$096 (schreiben) und \$02 (lesen)

Bit	Name	Funktion (wenn gesetzt)
15	SET/CLR	Bits setzen/löschen
14	BBUSY	Blitter arbeitet gerade (nur lesen)
13	BZERO	Ergebnis sämtlicher Blitteroperationen war 0 (nur lesen)
12 und 11		unbelegt
10	BLTPRI	Blitter-DMA hat Priorität über Prozessor
9	DMAEN	Gesamt-DMA einschalten (für Bits 0 bis 8)
8	BPLEN	Bit-Plane DMA einschalten
7	COPEN	Copper-DMA einschalten
6	BLTEN	Blitter-DMA einschalten
5	SPREN	Sprite-DMA einschalten
4	DSKEN	Disk-DMA einschalten
3-0	AUDxEN	Audio-DMA für Tonkanal x einschalten (Die Bitnummer entspricht der Nummer des Tonkanals)

Das DMACON-Register wird nicht wie ein normales Register beschrieben. Man kann immer nur Bits setzen oder Bits löschen. Dies wird durch Bit 15 in dem Datenwort festgelegt, das man ins DMACON-Register schreibt. Ist dieses Bit auf 1, werden alle gesetzten Bits des Datenworts auch im DMACON-Register gesetzt. Ist Bit 15 dagegen 0, werden alle gesetzten Bits im DMACON-Register gelöscht. Die übrigen Bits von DMACON bleiben immer unbeeinflusst.

Das mit DMAEN bezeichnete Bit 9 dient quasi als Hauptschalter. Ist es auf 0, sind alle DMA-Kanäle inaktiv, ungeachtet der Bits 0 bis 8. Soll ein DMA erlaubt werden, müssen das Bit des entsprechenden DMA-Kanals und das DMAEN-Bit gesetzt werden. Dazu folgendes Beispiel:

Es sei nur der Bitplane-DMA eingeschaltet (BPLEN = 1), aber ohne DMAEN-Bit. Der Wert des DMACON-Register beträgt also \$0100. Jetzt soll der Disk-DMA erlaubt werden. DSKEN und DMAEN müssen gesetzt und BPLEN gelöscht werden.

MOVE.W #\$0100,\$DFF096 ; löscht das BPLEN-Bit (SET/CLR = 0)

MOVE.W #\$8210,\$DFF096 ; setzt DSKEN und DMAEN (SET/CLR = 1)

Das DMACON-Register enthält jetzt den gewünschten Wert von \$0210.

Die Bits 13 und 14 können nur gelesen werden. Sie geben Auskunft über bestimmte Zustände des Blitters, näheres darüber im Blitter-Kapitel.

Bit 10 steuert die Priorität des Blitters über den Prozessor. Ist es gesetzt, hat der Blitter absolute Priorität über den 68000. Dies kann soweit gehen, daß der Prozessor während der gesamten Blitteroperation keinen einzigen Zugriff auf ein Chip-Register oder das Chip-RAM durchführen kann. Wenn es gelöscht ist, bekommt der Prozessor jeden vierten geraden Buszyklus vom Blitter. Dies verhindert, daß der Prozessor länger angehalten wird, wenn eine Betriebssystemroutine oder ein Programm im Fast RAM, die ja beide ungebremst laufen, einmal auf das Chip-RAM zugreifen müssen. Z.B. auf eine Datenstruktur des Betriebssystems oder auf einen der Ausnahmevektoren des 68000.

Die Abfrage der aktuellen Strahlposition

Da sich das gesamte DMA-Timing an der Position innerhalb einer Rasterzeile orientiert, möchte man manchmal wissen, an welcher Stelle der Zeile sich der Elektronenstrahl gerade befindet. Agnus besitzt dazu einen internen Zähler, der sowohl die horizontale als auch die vertikale Bildschirmposition enthält, nach der sich das gesamte System richtet. Zwei Register ermöglichen dem Prozessor den Zugriff auf diese Zähler:

VHPOS \$006 (lesen, VHPOSR) und \$02C (schreiben, VHPOSW)

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Funktion:	V7	V6	V5	V4	V3	V2	V1	V0	H8	H7	H6	H5	H4	H3	H2	H1

VPOS \$004 (lesen, VPOSR) und \$02A (schreiben, VPOSW)

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Funktion:	LOF	--	--	--	--	--	--	--	--	--	--	--	--	--	--	V8

Die mit H1 bis H8 bezeichneten Bits stellen die horizontale Strahlposition dar, sie entsprechen direkt den Nummern für die einzelnen Buszyklen in Abbildung 1.5.2.3 und haben damit eine Genauigkeit von zwei niedrig- oder vier hochauflösenden Punkten. Der Wert für die horizontale Position kann zwischen \$0 und \$E3 (0 bis 227) liegen. Die horizontale Austastlücke fällt in den Bereich von \$F bis \$35.

Die Bits für die vertikale Position, also die aktuelle Bildschirmzeile, sind auf zwei Register verteilt. Die unteren Bits V0 bis V7 liegen noch in VHPOS, das oberste Bit, V8, befindet sich in VPOS. Zusammen ergeben sie die Nummer der aktuellen Bildschirmzeile.

Es sind Zeilen von 0 bis 312 möglich. Die vertikale Austastlücke (der Bildschirm ist in diesem Bereich immer schwarz) reicht von Zeile 0 bis 25.

Das LOF-Bit (Long Frame) zeigt an, ob das gerade dargestellte Bild ein Long Frame (Halbbild mit ungeraden Zeilen) oder Short Frame (Halbbild mit geraden Zeilen) ist. Dieses Bit wird nur im Interlace-Modus benötigt. Normalerweise liegt es auf 1.

Man kann die Strahlposition auch setzen. Diese Möglichkeit wird aber kaum benötigt.

Eine weitere Funktion kommt den POS-Registern im Zusammenhang mit einem Lightpen zu. Wenn der Lightpen-Eingang von Agnus aktiviert ist (s. Kap. 1.5.5), und der Lightpen gegen den Bildschirm gehalten wird, enthalten sie seine Position. D.h. ihr Inhalt wird eingefroren, sobald der Lightpen durch den an

seiner Spitze vorbeifahrenden Elektronenstrahl ausgelöst wurde. Nach dem Ende der vertikalen Austastlücke, also ab Zeile 26, werden die Zähler wieder freigegeben. Will man die Lightpen-Position lesen, muß man folgendermaßen vorgehen:

- Auf Zeile 0 (Beginn der vertikalen Austastlücke) warten. Dies kann man am einfachsten mittels des Vertical Blanking Interrupts erreichen (s. nächstes Kapitel)
- Die beiden Zählerregister lesen.

Liegt die vertikale Position zwischen 0 und 25, also innerhalb der vertikalen Austastlücke, wurde kein Lightpen-Signal empfangen. Liegt der Wert dagegen außerhalb, stellt er die Position des Lightpens dar.

Zum Abschluß dieses Kapitels noch einige Details zu den Refresh-Zyklen:

Agnus besitzt einen integrierten 8-Bit-Refresh-Zähler. Er läßt sich über die Registeradresse \$28 beschreiben (Vorsicht! Dabei kann der Speicherinhalt verloren gehen!!!). Zu Beginn jeder Rasterzeile legt Agnus vier Refresh-Adressen auf den Chip-RAM-Adreßbus. D.h. der Inhalt jeder Speicherzeile wird alle 4 Millisekunden einmal aufgefrischt.

Während die Zeilenadresse auf dem Chip-RAM-Adreßbus ausgegeben wird, legt Agnus die Adressen bestimmter Strobe-Register auf den Registeradreßbus. Diese Strobe-Signale dienen dazu, den anderen Chips, Denise und Paula, den Beginn einer Rasterzeile oder eines Bildes mitzuteilen. Dies ist notwendig, da sich die Zähler für die Bildschirmposition innerhalb von Agnus befinden und keine Leitungen zur Übermittlung der Synchronisationssignale an die anderen Chips existieren. Im einzelnen gibt es vier verschiedene Strobe-Adressen:

Adr	Chip	Funktion
\$38	D	Vertikale Austastlücke eines Shortframes.
\$3A	D	Vertikale Austastlücke.
\$3C	D P	Diese Strobe-Adresse wird in jeder Rasterzeile außerhalb der vertikalen Austastlücke erzeugt.
\$3E	D	Kennzeichnung einer langen Rasterzeile (228 Zyklen)

Während des ersten Refresh-Zyklus wird immer eine der drei oberen Strobe-Adressen angesprochen. Normalerweise \$3C, innerhalb der vertikalen Austastlücke \$38 oder \$3A, je nach dem, ob es sich um ein Short- oder Longframe handelt.

Mit der vierten Adresse hat es folgende Bewandnis: Eine Rasterzeile hat rein rechnerisch eine Länge von 227.5 Buszyklen. Da es aber keine halben Zyklen gibt, wechseln Zeilen mit 227 und 228 Buszyklen einander ab. Die Strobe-Adresse \$3E signalisiert die 228 Zyklen langen Zeilen und wird während des zweiten Refresh-Zyklus erzeugt.

1.5.3 Interrupts

Fast alle Ein-/Ausgabeeinheiten der Customchips und die beiden CIAs sind in der Lage, einen Interrupt auszulösen. Eine spezielle Schaltung innerhalb Paulas übernimmt die Verwaltung der einzelnen Interrupt-Quellen und erzeugt daraus die Interrupt-Signale für den 68000. Dabei werden die sogenannten Autovektor-Interrupts des Prozessors benutzt. Und zwar die der Ebenen 0 bis 6. Der nicht maskierbare Interrupt (NMI) der Ebene 7 ist nicht vorgesehen. Die beiden Register sind das Interrupt-Anforderungs-Register (INTREQ, Interrupt-Request) und das Interrupt-Masken-Register (INTENA, Interrupt-Enable). Die Belegung der einzelnen Bits ist in beiden Registern identisch.

Interrupt-Enable- und Interrupt-Request-Registerbelegung

Registeradressen:	INTREQ	= \$09C (schreiben)
	INTREQR	= \$01E (lesen)
	INTENA	= \$09A (schreiben)
	INTENAR	= \$01C (lesen)

Bit	Name	IE	Funktion
15	SET/CLR		schreiben/lesen (siehe DMACON-Register)
14	INTEN	(6)	Interrupts erlauben
13	EXTER	6	Interrupt von CIA-B oder Expansion-Port
12	DSKSYN	5	Disk-Synchronisationswert erkannt
11	RBF	5	Eingabepuffer des Seriellen Ports voll
10	AUD3	4	Audiodaten Kanal 3 ausgegeben
9	AUD2	4	Audiodaten Kanal 2 ausgegeben
8	AUD1	4	Audiodaten Kanal 1 ausgegeben
7	AUD0	4	Audiodaten Kanal 0 ausgegeben
6	BLIT	3	Blitter fertig
5	VERTB	3	Beginn der vertikalen Austastlücke erreicht
4	COPER	3	Reserviert für Copper-Interrupts
3	PORTS	2	Interrupt von CIA-A oder Expansion-Port
2	SOFT	1	Reserviert für Software-Interrupts
1	DSKBLK	1	Disk-DMA-Transfer beendet
0	TBE	1	Ausgabepuffer des seriellen Ports leer

Die unteren dreizehn Bits stehen für die einzelnen Interrupt-Quellen. Die CIA-Interrupts wurden je zu einem einzigen Interrupt zusammengefaßt. Die Bits im DMAREQ-Register informieren darüber, welche Interrupts aufgetreten sind. Es wird in diesem Fall das zugehörige Bit gesetzt. Um einen Prozessor-Interrupt auszulösen, müssen das korrespondierende Bit im DMAENA-Register und auch das INTEN-Bit gesetzt sein. Das INTEN-Bit agiert dabei als Hauptschalter für die übrigen 14 Interrupt-Quellen, die mit den einzelnen Bits des INTENA-Registers getrennt ein- und ausgeschaltet werden können. Nur

wenn INTEN auf 1 liegt, können überhaupt Interrupts ausgelöst werden.

Sind sowohl das INTEN-Bit als auch zwei zusammengehörige Bits im INTENA- und INTREQ-Register gesetzt, wird ein Prozessor-Interrupt ausgelöst. Die entsprechenden Autovektornummern befinden sich in der mit IE (Interrupt-Ebene) bezeichneten Spalte der Tabelle. Zur Erinnerung hier noch einmal die Adressen der 7 Interrupt-Autovektoren:

Vektor Nr.	Adresse (Dez/Hex)	Autovektor Ebene
25	100 / \$64	Autovektor Ebene 1
26	104 / \$68	Autovektor Ebene 2
27	108 / \$6C	Autovektor Ebene 3
28	112 / \$70	Autovektor Ebene 4
29	116 / \$74	Autovektor Ebene 5
30	120 / \$78	Autovektor Ebene 6
(31	124 / \$7C	Autovektor Ebene 7)

Wie man sieht, wurden den Interrupts, die eine schnellere Bearbeitung erfordern, höhere Interrupt-Ebenen verliehen.

Um die Bits in den beiden Registern zu ändern muß man, wie schon beim DMACON-Register beschrieben (1.5.2), mit einem SET/CLR-Bit arbeiten.

Nach der Bearbeitung eines Interrupts muß das auslösende Bit im INTREQ-Register vom Prozessor zurückgesetzt werden. Im Gegensatz zu den Interrupt-Kontrollregistern der CIAs werden die Bits des INTREQ-Registers nicht automatisch beim Lesen gelöscht.

Wenn man ein Bit im INTREQ-Register mittels eines MOVE-Befehls setzt, hat dies dieselbe Wirkung, als wenn der entsprechende Interrupt aufgetreten wäre. Auf diese Weise wird z.B. der Software-Interrupt (SOFT, Bit 2) erzeugt. Auch der Copper kann nur durch Schreiben in INTREQ seinen Interrupt erzeugen.

Eine Besonderheit ist das Bit 14 im INTREQ-Register, es hat dort keine spezifische Funktion wie in INTENA. Aber wenn

man es durch Schreiben in INTREQ setzt und sich INTEN im INTENA-Register auf High befindet, wird ein Interrupt der Ebene 6 erzeugt.

Bei jedem Interrupt von CIA-A wird Bit 3 im DMAREQ-Register gesetzt. Für CIA-B ist es Bit 13. Die Interrupt-Quelle in dem entsprechenden CIA muß durch Lesen des Interrupt-Kontrollregisters des CIAs ermittelt werden.

Die Interrupts Nr. 3 und 13 können auch durch Erweiterungskarten am Expansion-Port ausgelöst werden.

Das Interruptbit 5 zeigt den sogenannten Vertical Blank Interrupt an. Dieser tritt immer zum Beginn jedes Halbbilds auf, am Start der vertikalen Austastlücke (Zeile 0) und damit 50mal in der Sekunde.

Die übrigen Interrupts werden in den dazugehörigen Kapiteln behandelt.

1.5.4 Der Coprozessor Copper

Der Copper ist ein einfacher Coprozessor. Er hat die Aufgabe, die verschiedenen Register der Customchips zu festgelegten Zeitpunkten automatisch mit bestimmten Werten zu beschreiben. Genauer gesagt ist der Copper in der Lage, an beliebigen Bildschirmpositionen die Inhalte einiger Register zu verändern. Er kann dadurch den Bildschirm in unterschiedliche Bereiche aufteilen, die dann verschiedene Farben und Auflösungen haben können. Diese Fähigkeit wird z.B. bei der Verwendung mehrerer Screens benutzt.

Der Copper wird deshalb als Coprozessor bezeichnet, weil er, wie ein richtiger Prozessor auch, über ein Programm verfügt, das sich im Speicher befindet und von ihm Befehl für Befehl abgearbeitet wird. Allerdings kennt der Copper nur drei verschiedene Befehle, mit denen man aber eine Menge anfangen kann:

MOVE

Der Move-Befehl schreibt einen unmittelbaren Wert in ein beliebiges Customchip-Register.

WAIT

Der Wait-Befehl wartet darauf, daß der Elektronenstrahl eine bestimmte Bildschirmposition erreicht.

SKIP

Der Skip-Befehl überspringt den nächsten Befehl, wenn der Elektronenstrahl eine festgelegte Bildschirmposition schon erreicht hat. Mit diesem Befehl lassen sich bedingte Verzweigungen programmieren.

Das Programm für den Copper nennt man Copperlist. In ihr liegen die Befehle direkt hintereinander, wobei jeder Befehl immer aus zwei Wörtern besteht. Beispiel:

```
Wait (X1,Y1) ;Wartet, bis die Bildschirmposition X1,Y1 erreicht ist
Move #0,$180 ;Schreibt den Wert 0 in das Hintergrundfarbregister
Move #9,$181 ;Schreibt den Wert 1 in das Farbregister 1
Wait (X2,Y2) ;Wartet, bis die Bildschirmposition X2,Y2 erreicht ist
...          usw.
```

Zum Betrieb des Coppers reicht die Copperlist alleine nicht aus. Es sind noch einige Register notwendig, die die für den Copper notwendigen Parameter enthalten.

Die Copperregister:

Reg.	Name	Funktion
\$080	COP1LCH	Diese beiden Register enthalten gemeinsam die
\$082	COP1LCL	18-Bit-Adresse der ersten Copperlist.
\$084	COP2LCH	Diese beiden Register enthalten gemeinsam die
\$086	COP2LCL	18-Bit-Adresse der zweiten Copperlist
\$088	COPJMP1	Laden der Adresse der ersten Copperlist in den Programm- zähler des Coppers
\$08A	COPJMP2	Laden der Adresse der zweiten Copperlist in den Pro- grammzähler des Coppers
\$02E	COPCON	Dieses Register enthält nur ein einziges Bit (Bit 0). Ist es gesetzt, kann der Copper auch auf die Register von \$040 bis \$7E zugreifen. (Dies sind die zum Blitter gehörenden Register.)

Sämtliche Copperregister sind Nur-Schreib-Register.

Die beiden COPxLC-Register enthalten jeweils die Adresse einer Copperlist. Da diese Adresse 19 Bit lang ist, werden zwei Register pro Adresse benötigt. Sie können, wie in Kapitel 1.5.2 besprochen, mit einem MOVE.L-Befehl in das erste Register gemeinsam beschrieben werden. Die Copperlist muß, wie alle sonstigen Daten für die Customchips, innerhalb der 512 KB Chip-RAM liegen.

Der Copper benutzt einen internen Programmzähler als Zeiger auf den aktuellen Befehl. Er wird mit jedem abgearbeiteten Befehl um zwei Worte erhöht. Um den Copper jetzt bei einer bestimmten Adresse beginnen lassen zu können, muß die Anfangsadresse der Copperlist in den Programmzähler übertragen werden. Dazu dienen die beiden COPJMPx-Register. Sie stellen sogenannte Strobe-Register da, d.h. ein Wert, den man in eines dieser Register schreibt, wird nicht gespeichert, sondern dient lediglich als Auslöser für eine einmalige Aktion. Aus diesem Grund ist der Wert auch völlig gleichgültig, es kommt nur auf den Zugriff auf ein solches Register an.

Beim Copper dienen diese beiden Register dazu, den Inhalt des entsprechenden COPxLC-Registers in den Programmzähler zu übertragen. Schreibt man also in COPJMP1, wird die Adresse in

COP1LC in den Programmzähler übertragen, was zur Folge hat, das der Copper die Ausführung seines Programms dort fortsetzt. Gleiches gilt für COPJMP2 und COP2LC.

Am Beginn der vertikalen Austastlücke, in Zeile 0, wird der Programmzähler automatisch mit dem Wert von COP1LC geladen. Dies hat zur Folge, das der Copper in jedem Bild das selbe Programm ausführt.

Der Aufbau der Befehle

Bit	MOVE		WAIT		SKIP	
	BW1	BW2	BW1	BW2	BW1	BW2
15	x	DW15	VP7	BFD	VP7	BFD
14	x	DW14	VP6	VM6	VP6	VM6
13	x	DW13	VP5	VM5	VP5	VM5
12	x	DW12	VP4	VM4	VP4	VM4
11	x	DW11	VP3	VM3	VP3	VM3
10	x	DW10	VP2	VM2	VP2	VM2
9	x	DW9	VP1	VM1	VP1	VM1
8	RA8	DW8	VP0	VM0	VP0	VM0
7	RA7	DW7	HP8	HM8	HP8	HM8
6	RA6	DW6	HP7	HM7	HP7	HM7
5	RA5	DW5	HP6	HM6	HP6	HM6
4	RA4	DW4	HP5	HM5	HP5	HM5
3	RA3	DW3	HP4	HM4	HP4	HM4
2	RA2	DW2	HP3	HM3	HP3	HM3
1	RA1	DW1	HP2	HM2	HP2	HM2
0	0	DW0	1	0	1	1

Legende: x Dieses Bit ist unbenutzt. Sollte mit 0 initialisiert werden.

RA Registeradresse
 DW Datenwort
 VP Vertikale Strahlposition
 VM Vertikale Maskenbits
 HP Horizontale Strahlposition
 HM Horizontale Maskenbits
 BFD Blitter Finish Disable

Der MOVE-Befehl

Der MOVE-Befehl wird gekennzeichnet durch eine 0 in Bit 0 des ersten Befehlsworts. Mittels dieses Befehls ist es möglich, ein CustomChip-Register mit einem unmittelbaren Wert zu beschreiben. Die Registeradresse des gewünschten Registers kommt in die unteren 9 Bits des ersten Datenworts. Dabei muß Bit 0 immer 0 bleiben (Ist bei den Registeradressen sowieso schon auf 0, da die Register ausschließlich auf geraden Adressen liegen). Das zweite Befehlswort enthält das Datenbyte, das in das Register geschrieben werden soll.

Bei der Registeradresse gibt es einige Einschränkungen. Normalerweise kann der Blitter die Register im Bereich von \$000 bis \$07F nicht beeinflussen. Setzt man das unterste (und auch einzige) Bit im COPCON-Register, ist es dem Copper auch möglich, in die Register von \$040 bis \$07F zu schreiben. Dadurch kann der Copper dann den Blitter benutzen. Ein Zugriff auf die untersten Register (\$000 bis \$03F) ist allerdings immer verboten.

Der WAIT-Befehl

Der WAIT-Befehl wird durch eine 1 in Bit 0 des ersten und eine 0 in Bit 0 des zweiten Befehlsworts gekennzeichnet. Er veranlaßt den Blitter, mit der weiteren Befehlsausführung bis zum Erreichen der gewünschten Strahlposition zu warten. Ist sie beim Auftreten eines Wait-Befehls schon größer als die im Befehl angegebene, der Strahl also schon an der gewünschten Position vorbei, macht der Copper sofort mit der nächsten Instruktion weiter.

Diese Position läßt sich getrennt für die vertikalen Zeilen und horizontalen Spalten einstellen. Vertikal beträgt die Auflösung eine Rasterzeile. Da nur acht Bit für die vertikale Position vorgesehen sind, es aber 313 Zeilen gibt, kann der Wait-Befehl nicht zwischen den ersten 256 und den restlichen 57 Zeilen unterscheiden. Die untersten 8 Bits (mehr lassen sich im Wait-Be-

fehl nicht angeben) sind z.B. sowohl in Zeile 0 und Zeile 256 gleich. Will man gezielt auf eine Zeile im unteren Bereich warten, muß man sich mit zwei WAIT-Befehlen behelfen.

1. WAIT auf Zeile 255
2. WAIT auf gewünschte Zeile, unter Vernachlässigung des 9. Bits

Horizontal gibt es 112 mögliche Positionen, da die beiden unteren Bits der horizontalen Position, HP0 und HP1, nicht angegeben werden können. Das Befehlswort des MOVE-Befehls enthält ja nur die Bits HP2 bis HP8. D.h. die horizontale Koordinate eines WAIT-Befehls läßt sich nur in Schritten von vier niedrig auflösenden Punkten angeben.

Das zweite Befehlswort enthält die sogenannten Maskenbits. Mit ihnen kann man festlegen, welche Bits der horizontalen und vertikalen Position überhaupt zum Vergleich mit der aktuellen Strahlposition herangezogen werden. Nur die Positionsbits, deren zugehörige Maskenbits gesetzt sind, werden beachtet. Dies eröffnet vielfältige Möglichkeiten:

Wait vertikale Position \$0F und vertikale Maske \$0F

bewirkt, daß alle 16 Zeilen die Wait-Bedingung erfüllt wird, nämlich immer, wenn die unteren 4 Bits auf 1 sind, da Bits 4 bis 6 nicht mehr in den Vergleich mit einbezogen werden (Maskenbits 4 bis 6 sind auf 0). Das 7. Bit der vertikalen Position läßt sich nicht maskieren. Aus diesem Grund funktioniert das obige Beispiel nur im Bereich der Zeilen 0 bis 127 und 256 bis 313.

Das BFD (Blitter Finish Disable) Bit hat folgende Funktion: Soll der Copper dazu benutzt werden, eine Blitter-Operation zu starten, muß er wissen, wann der Blitter mit der vorangegangenen fertig ist. Ist das BFD-Bit gelöscht, wartet der Copper bei jedem Wait-Befehl, bis der Blitter seine Operation beendet hat. Erst dann wird die übrige Wait-Bedingung geprüft. Dies kann

man verhindern, indem man das BFD-Bit setzt. Dadurch ignoriert der Copper den aktuellen Blitter-Status. Wenn der Copper keine Blitter-Register beeinflussen soll, setzt man dieses Bit daher auf 1.

Der SKIP-Befehl

Der SKIP-Befehl ist in seinem Aufbau mit dem Wait-Befehl identisch. Lediglich Bit 0 im zweiten Befehlswort ist gesetzt, um den Skip-Befehl vom Wait-Befehl zu unterscheiden. Der Skip-Befehl prüft, ob die tatsächliche Strahlposition größer oder gleich der im Befehlswort festgelegten ist. Fällt dieser Vergleich positiv aus, überspringt der Copper den nächsten Befehl. Sonst fährt er in seiner Programmabarbeitung sofort mit dem darauffolgenden Befehl fort. Der Skip-Befehl ermöglicht damit den Aufbau bedingter Verzweigungen. So kann der auf Skip folgende Befehl ein Move in eines der COPJMP-Register sein, wodurch dann je nach Strahlposition ein Sprung ausgelöst werden würde.

Der Aufbau einer Copperlist

Eine einfache Copperlist besteht aus einer Abfolge von Wait- und Move-, seltener auch Skip-Befehlen. Ihre Anfangsadresse befindet sich in COPLC1. Um die Copperlist zu beenden, muß zu einem Trick gegriffen werden. Nach der letzten Instruktion folgt noch ein Wait-Befehl, der aber eine unmögliche Strahlposition als Bedingung hat. Dadurch wird die Abarbeitung der Copperlist beendet, bis durch den Anfang eines neuen Bildes wieder die Adresse von COPLC1 in den Programmzähler des Coppers geladen und die Copperlist erneut bearbeitet wird. Wait (\$0,\$fe) erfüllt diese Bedingung, denn eine horizontale Position größer \$E4 ist nicht möglich.

Der Copper-Interrupt

Wie man weiß, gibt es in den Interrupt-Registern ein spezielles Bit für den Copper-Interrupt. Diesen Interrupt kann man mit einem Move-Befehl in das INTREQ-Register auslösen:

```
MOVE #$8010,INTREQ ;SET/CLR und COPER setzen
```

Genauso könnte man auch jedes andere Bit dieses Registers beeinflussen, aber Bit 4 ist speziell für den Copper vorgesehen.

Ein Copper-Interrupt kann dazu dienen, dem Prozessor das Erreichen einer bestimmten Bildposition mitzuteilen. Dadurch lassen sich sogenannte Raster-Interrupts programmieren, d.h. die Unterbrechung des Prozessors in einer bestimmten Bildschirmzeile (und Spalte).

Der Copper-DMA

Der Copper holt seine Befehle über einen eigenen DMA-Kanal aus dem Speicher. Er belegt die geraden Buszyklen und hat dabei Vorrang vor Blitter und 68000. Jeder Befehl benötigt zwei Zyklen, da ja zwei Befehlsworte gelesen werden müssen. Der Wait-Befehl benötigt noch einen zusätzlichen Zyklus beim Erreichen der gewünschten Strahlposition. Während der Wartephase eines Wait-Befehls gibt der Copper den Bus frei.

Das COPEN-Bit im DMACON-Register dient dazu, den Copper-DMA ein- und auszuschalten. Löscht man dieses Bit, gibt der Copper den Bus frei und führt keine weiteren Befehle aus. Setzt man es, beginnt er seine Programmausführung bei der Adresse in seinem Programmzähler. Es ist daher unbedingt notwendig, vor dem Einschalten des Copper-DMA für eine verlässliche Adresse zu sorgen. Ein in ungewissen Speicherbereichen laufender Copper kann das System zum Absturz bringen. Die übliche Initialisierungssequenz für den Copper sieht daher folgendermaßen aus:

```

LEA $DFF000,A5 ;Basisadresse der Register nach A5
MOVE.W #$0080,DMACON(A5) ;Copper-DMA aus
MOVE.L #Copperlist,COP1LCH(A5);Adresse der Copperlist setzen
COPJMP1(A5);Diese Adresse in Programmzähler des Coppers übertragen
MOVE.W #$8080,DMACON(A5) ;Copper-DMA wieder einschalten

```

Beispielprogramm

Zum Abschluß noch ein Beispielprogramm. Es bringt mit Hilfe von zwei WAIT- und drei MOVE-Befehlen die deutsche Fahne auf den Bildschirm. Sie läßt sich schon mit einer einfachen Copperlist erzeugen und eignet sich daher gut als Beispiel. Geben Sie das Programm mit einem der handelsüblichen Assembler für den Amiga ein (z.B. Profimat Amiga):

```

;*** Beispiel für eine einfache Copperlist ***

;CustomChip-Register

INTENA = $9A ;Interrupt-Enable-Register (schreiben)
DMACON = $96 ;DMA-Kontrollregister (schreiben)
COLOR00 = $180 ;Farbpalettenregister 0

;Copper Register

COP1LC = $80 ;Adresse der 1. Copperlist
COP2LC = $84 ;Adresse der 2. Copperlist
COPJMP1 = $88 ;Sprung nach Copperlist 1
COPJMP2 = $8a ;Sprung nach Copperlist 2

;CIA-A Portregister A (Maus Taste)

CIAAPRA = $BFE001

;Exec Library Base Offsets

OpenLibrary = -30-522 ;LibName,Version/a1,d0
Forbid = -30-102
Permit = -30-108
AllocMem = -30-168 ;ByteSize,Requirements/d0,d1
FreeMem = -30-180 ;MemoryBlock,ByteSize/a1,d0

;graphics base

```

```
StartList = 38

;sonstige Labels

Execbase = 4
Chip = 2 ;Chip-RAM anfordern

;*** Vorprogramm ***

;Speicher fuer Copperlist anfordern

Start:
move.l Execbase,a6
moveq #CLsize,d0 ;Parameter für AllocMem setzen
moveq #chip,d1 ;Chip-RAM verlangen
jsr AllocMem(a6) ;Speicher anfordern
move.l d0,CLadr ;Adresse des RAM-Bereichs speichern
beq.s Ende ;Fehler! -> Ende

;Copperlist nach CLadr kopieren

lea CLstart,a0
move.l CLadr,a1
moveq #CLsize-1,d0 ;Schleifenzähler setzen
CLcopy:
move.b (a0)+,(a1)+ ;Copperlist Byte für Byte kopieren
dbf d0,CLcopy

;*** Hauptprogramm ***

jsr forbid(a6) ;Task Switching sperren
lea $dff000,a5 ;Basisadresse der Register nach A5
move.w #$03a0,dmacon(a5) ;DMA sperren
move.l CLadr,cop1lc(a5) ;Adresse der Copperlist nach COP1LC
clr.w copjmp1(a5) ;In Programmzähler des Coppers laden

;CopperDMA einschalten

move.w #$8280,dmacon(a5)

;Auf linke Maustaste warten

Wait: btst #6,ciaapra ;Bit testen
bne.s Wait ;gesetzt? Dann warten.
```

;*** Nachprogramm ***

;Alte Copperlist wieder aktivieren

```
move.l #GRname,a1 ;Parameter fuer OpenLibrary setzen
clr.l d0
jsr OpenLibrary(a6) ;Graphics Library öffnen
move.l d0,a4 ;Adresse von GraphicsBase nach a4
move.l StartList(a4),cop1lc(a5) ;Adresse der Startlist laden
clr.w copjmp1(a5)
move.w #$83e0,dmacon(a5) ;alle nötigen DMA-Kanäle ein
jsr permit(a6) ;Task-Switching erlauben
```

;Speicher fuer Copperlist wieder freigeben

```
move.l CLadr,a1 ;Parameter fuer FreeMem setzen
moveq #CLsize,d0
jsr FreeMem(a6) ;Speicher wieder freigeben
Ende:
clr.l d0 ;Fehlerflag löschen
rts ;Programm verlassen
```

;Variablen

CLadr: dc.l 0

;Konstanten

GRname: dc.b "graphics.library",0
even

;Copperlist

```
CLstart:
dc.w color00,$0000 ;Hintergrundfarbe schwarz
dc.w $780f,$fffe ;Auf Zeile 120 warten
dc.w color00,$0f00 ;Auf Rot umschalten
dc.w $d70f,$fffe ;Zeile 215
dc.w color00,$0fb0 ;Gold
dc.w $ffff,$fffe ;unmögliche Position: Ende der Copperlist
CLend:
```

CLsize = CLend - CLstart

;Ende des Programms

Dieses Programm installiert die Copperlist und wartet dann, bis die linke Maustaste gedrückt wird. Leider ist es beim Amiga nicht leicht, dies so zu erledigen, daß man das Programm am Ende ohne einen Reset verlassen kann.

Als erstes benötigt man Speicher, in dem man die Copperlist unterbringen kann. Wie alle Daten für die Customchips muß sie im Chip-RAM stehen. Da man nicht sicher sein kann, ob sich das eigene Programm überhaupt im Chip-RAM befindet, ist es notwendig, die Copperlist in das Chip-RAM zu kopieren. In einem Multitasking-Betriebssystem wie das des Amiga kann man nicht einfach irgend etwas in den Speicher schreiben. Man muß den Speicher erst anfordern. Dies geschieht in dem Programm mittels der AllocMem-Routine. Diese gibt in D0 die Adresse des angeforderten Chip-RAM-Bereichs zurück, in den danach die Copperlist kopiert wird.

Als nächstes wird mit dem Aufruf von 'Forbid' das Task-Switching abgeschaltet, d.h. der Amiga bearbeitet ab sofort nur noch unser Programm. Dies verhindert, daß unser Programm von einem anderen gestört wird.

Jetzt endlich wird der Copper initialisiert und gestartet.

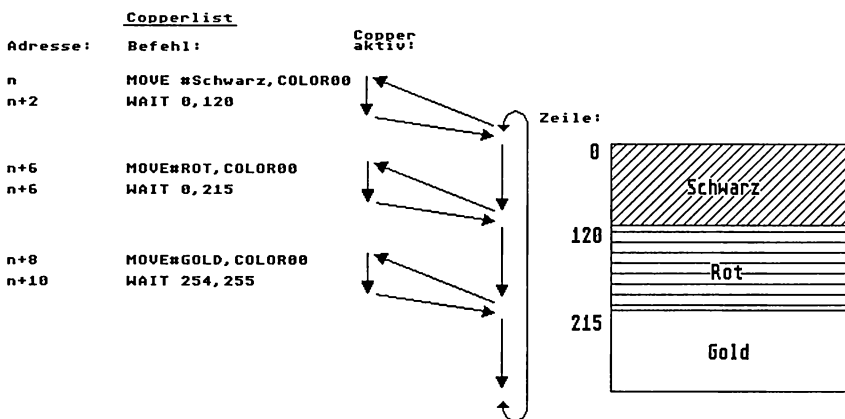
Danach testet das Programm die linke Maustaste, indem es das zugehörige Portbit von CIA-A abfragt (siehe Kapitel 1.2.2).

Drückt man die Maustaste, verläßt der Prozessor die Warteschleife.

Um wieder zu der alten Anzeige zurückzukommen, wird eine spezielle Copperlist in den Copper geladen und gestartet. Diese Copperlist heißt Startup-Copperlist und initialisiert den Bildschirm. Ihre Adresse findet man im Variablenbereich des für die Grafikfunktionen zuständigen Teils vom Betriebssystem. Zum Abschluß wird dann noch das Task-Switching mittels 'Permit' wieder eingeschaltet und der belegte Speicher mit FreeMem freigegeben.

Nun, dieses Programm überfällt Sie mit einer Vielzahl unbekannter Funktionen des Betriebssystems. Leider läßt sich dies nicht vermeiden, wenn das Programm ordnungsgemäß funktionieren soll. Aber es macht nichts, wenn Sie jetzt nicht alles verstehen. Es kommt ja hauptsächlich auf den Copper an, und dieser Teil des Programms müßte verständlich sein. In den späteren Kapiteln dieses Buches werden Sie ja in die Geheimnisse des Betriebssystems und seiner Routinen eingeführt. Tippen Sie also dieses Beispiel ab und experimentieren Sie mit der Copperlist. Ändern Sie die WAIT-Befehle, oder fügen Sie, ganz nach Belieben, neue hinzu. Wer es sich zutraut, kann es ja auch einmal mit einem SKIP-Befehl versuchen.

Noch ein Hinweis zur Copperlist: Die beiden WAIT-Befehle enthalten als horizontale Position jeweils \$E. Dies ist der Anfang der horizontalen Austastlücke. Dadurch vollzieht der Copper die Farbumschaltung außerhalb des sichtbaren Bereichs. Setzt man 0 als horizontale Position, kann man die Farbumschaltung am äußersten rechten Rand des Bildschirms beobachten.



Ablauf der Copperlist aus unserem Beispiel

Abb. 1.5.4

1.5.5 Playfields

Die Bildschirmausgabe des Amiga besteht aus zwei Grundelementen: Sprites und Playfields. In diesem Kapitel sollen Aufbau und Programmierung sämtlicher Arten von Playfields besprochen werden. Die Sprites folgen dann in Kapitel 1.5.6.

Das Playfield ist die Grundlage der normalen Bildschirmdarstellung. Es besteht aus mindestens einer und maximal sechs Bitplanes. (Der Aufbau einer Bitplane wurde in 1.5.2 ja schon erläutert.) Ein Playfield stellt also einen Grafikbildschirm dar, der aus einer variablen Anzahl einzelner Speicherbereiche, den Bitplanes, zusammengesetzt ist. Der Amiga bietet eine große Anzahl unterschiedlicher Möglichkeiten, Playfields darzustellen:

- Zwischen 2 und 4096 Farben gleichzeitig in einem Bild!
- Auflösungen von 16 auf 1 bis zu 704 auf 625 Punkte!
- Zwei von einander völlig unabhängige Playfields möglich
- Ruckfreies Scrolling in beide Richtungen (Smooth-Scrolling)

All diese Möglichkeiten kann man in zwei Gruppen aufteilen.

1. Die Kombination der Bitplanes zur Errechnung der Farbe der einzelnen Bildschirmpunkte (die Darstellung des Bitmusters aus den Bitplanes auf dem Bildschirm).
2. Die Festlegung von Form, Größe und Position des/der Playfields (Aufbau der Playfields).

Die verschiedenen Darstellungsmöglichkeiten

Durch die Verwendung von 1 bis 6 Bitplanes wird jeder Punkt von ebenso vielen Bits repräsentiert. Dieser Wert muß nun in eine von 4096 Farben umgewandelt werden, da jedes Pixel auf dem Bildschirm natürlich nur eine Farbe haben kann.

Der Amiga erzeugt seine Farben durch Mischen der drei Grundfarben Rot, Grün und Blau. Jede dieser drei Komponenten kann 16 verschiedene Intensitätsstufen annehmen. Dadurch entstehen insgesamt 4096 Farbtöne ($16 \cdot 16 \cdot 16 = 4096$). Zur Speicherung der Farbwerte werden pro Komponente 4 Bit benötigt. Dies macht 12 Bit pro Farbe.

Wollte man für jeden Punkt eine von 4096 Farben zulassen, bräuhete man 12 Bit pro Punkt. Es sind aber maximal 6 Bit pro Punkt möglich. Aus diesem Grund müssen die 6 Bit umgerechnet werden, um für den sichtbaren Punkt eine der 4096 möglichen Farben zu erhalten.

Die Farbpalette

1. Farbwahl über Farbtabelle

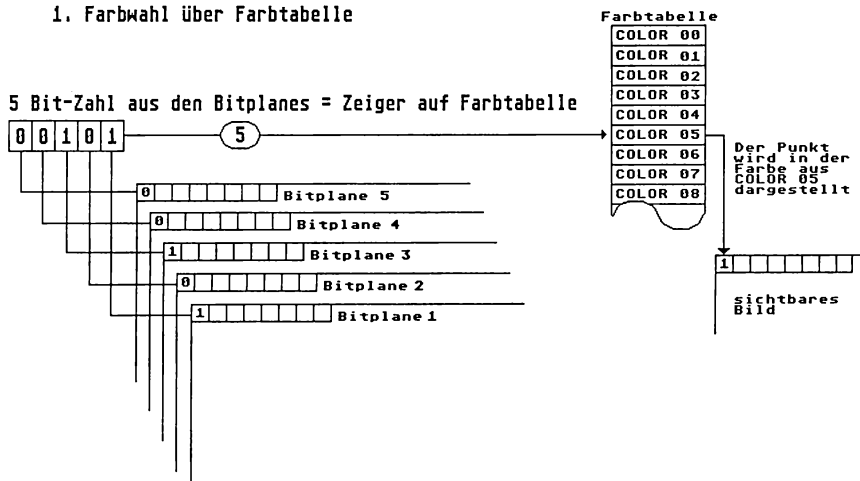


Abb. 1.5.5.1

Man verwendet dazu eine sogenannte Farbpalette oder Farbtabelle. Diese enthält beim Amiga 32 Einträge, die je einen 12 Bit Farbwert aufnehmen können. Der Wert des ersten Farbregisters COLOR00 dient gleichzeitig als Hintergrund und Rahmenfarbe.

Die Farbpalettenregister 0 bis 31 (COLOR00 bis COLOR31 (nur Schreiben möglich)):

Registeradr.	Farbpalettenregister
\$180	COLOR00
\$182	COLOR01
...	usw.
\$1BE	COLOR31

Aufbau eines Tabellenelements:

Bit:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
COLORxx:	x	x	x	x	R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0

R0-R3 Vier-Bit-Wert für den Rot-Anteil

G0-G3 Vier-Bit-Wert für den Grün-Anteil

B0-B3 Vier-Bit-Wert für den Blau-Anteil

Die oberen vier mit "x" bezeichneten Bits sind ungenutzt.

Der aus den Bit-Planes gewonnene Wert wird nun als Zeiger auf ein Tabellenelement verwendet. Da es nur 32 dieser Farbtabelregister gibt, können in diesem Modus maximal 5 Bitplanes miteinander kombiniert werden. Das Bit aus der Bitplane mit der niedrigsten Nummer liefert das LSB dieses Eintrags, die Bitplane mit der höchsten Nummer das MSB.

Diese Art der Farbgewinnung über eine Tabelle ermöglicht maximal 32 Farben in einem Bild, die aber aus der Gesamtanzahl von 4096 gewählt werden können. Im hochauflösenden Modus dürfen insgesamt nur 4 Planes gleichzeitig aktiv sein. Hier sind dann 16 Farben die Grenze. Es ist bei dieser Darstellungsart egal, wie viele Planes miteinander kombiniert werden. Es bleiben dann eben einige Farbregister unbenutzt:

Anzahl der Bitplanes	Farben	benutzte Farbregister
1	2	COLOR00 - COLOR01
2	4	COLOR00 - COLOR03
3	8	COLOR00 - COLOR07
4	16	COLOR00 - COLOR15
5	32	COLOR00 - COLOR31

Der Extra-Halfbright-Modus

Im niedrigauflösenden Modus können maximal 6 Bitplanes verwendet werden. Dies ergibt einen Wertebereich von 2^6 oder 0 bis 63. Es sind aber nur 32 Farbregister vorhanden. Man verwendet aus diesem Grund dafür eine spezielle Technik, den Extra-Halfbright-Modus. Die unteren 5 Bits (Bits 0 bis 4 aus den Planes 1 bis 5) dienen weiterhin als Zeiger auf ein Farbregister. Der Inhalt dieses Farbregisters wird direkt auf den Bildschirm ausgegeben, wenn Bit 5 (aus Bitplane 6) = 0 ist. Liegt dieses Bit aber auf 1, wird der Farbwert durch zwei dividiert, bevor er auf dem Bildschirm ausgegeben wird.

Durch zwei dividiert bedeutet, daß die Werte der drei Farbkomponenten um 1 Bit nach rechts geschoben werden, was ja einer Division durch zwei entspricht. Da die einzelnen Komponenten danach nur noch halb so groß sind, wird auf dem Bildschirm zwar dieselbe Farbe dargestellt, aber mit halber Helligkeit, daher auch der englische Name Extra-Halfbright, was soviel heißt wie "Besonderer Modus für halbe Helligkeit".

Beispiel:

Bit Nr.: 5 4 3 2 1 0

Wert aus den Bitplanes: 1 0 0 1 0 0

ergibt Tabelleneintrag Nr. 8 (binär 00100 ist gleich 8)

COLOR08 soll folgenden Wert enthalten (Farbe: Orange):

```
R3 R2 R1 R0 G3 G2 G1 G0 B3 B2 B1 B0
1  1  1  0  0  1  1  0  0  0  0  1
```

Da Bit 5 = 1 ist, werden die Werte um 1 Bit verschoben:

R3	R2	R1	R0	G3	G2	G1	G0	B3	B2	B1	B0
0	1	1	1	0	0	1	1	0	0	0	0

Dieser Wert entspricht immer noch Orange, aber jetzt nur noch halb so hell.

Durch entsprechende Auswahl der Farbwerte in den 32 Registern ist es im Extra-Halfbright-Modus machbar, jedem Punkt eine von 64 möglichen Farben zu geben. In die Farbregister kommen die hellen Farben, die dazugehörigen dunkleren Töne werden durch Setzen von Bit 5 erreicht.

Der Hold-And-Modify-Modus

Dieser Modus ermöglicht die Darstellung aller 4096 Farben in einem Bild. Er ist wie der Extra-Halfbright-Modus nur bei niedriger Auflösung möglich, da er ebenfalls alle 6 Bitplanes benötigt. In diesem Modus macht man sich die Tatsache zunutze, daß die Farben in einem normalen Bild selten sprunghaft wechseln. Meistens benötigt man fließende Übergänge vom Hellen ins Dunkle oder umgekehrt.

Im Hold-And-Modify-Modus, kurz HAM genannt, wird die Farbe des vorangegangenen Punkts von dem darauffolgenden modifiziert. Dadurch kann man die gewünschten feinen Farb-abstufungen entstehen lassen, indem man z.B. den Blauanteil mit jedem Pixel um einen Schritt erhöht. Eingeschränkt ist man allerdings dadurch, daß immer nur eine Komponente beeinflußt werden kann, von einem Punkt zum nächsten läßt sich entweder der Rot-, Grün- oder Blau-Wert verändern, nie aber mehrere gleichzeitig. Um aber einen fließenden Übergang von Dunkel nach Hell zu erzielen, müssen bei vielen Mischfarben alle drei Farbanteile verändert werden. Dies kann im HAM-Modus nur erreicht werden, indem man mit jedem Punkt eine der Komponenten auf den gewünschten Wert setzt. Man benötigt dafür dann drei Punkte.

Als Ausgleich kann man die Farbe eines Pixels auch direkt ändern, indem man weiterhin eine von 16 Farben aus der Farbtabelle holen kann.

Wie wird der Wert aus den Bitplanes im HAM-Modus interpretiert?

Die oberen beiden Bits (Bits 4 und 5 aus Bitplanes 5 und 6) bestimmen die Verwendung der unteren vier Bits (Bitplanes 1 bis 4). Sind Bit 4 und 5 auf 0, werden die restlichen vier Bits wie üblich als Zeiger auf eines der Farbpalettenregister verwendet. Es können also 16 Farben direkt ausgewählt werden. Bei einer Kombinationen der Bits 4 und 5, die ungleich 0 ist, wird der Farbwert des letzten Punkts genommen (links von dem aktuellen Pixel), zwei der drei Farbkomponenten bleiben erhalten, die dritte wird durch den Wert aus den unteren vier Bits des aktuellen Punkts ersetzt. Die Wahl zwischen den drei Farbanteilen übernehmen dabei die oberen beiden Bits.

Dies hört sich komplizierter an, als es ist. Folgende Tabelle veranschaulicht die Verwendung der unterschiedlichen Bitkombinationen:

Bit Nr.:						Funktion
5	4	3	2	1	0	
0	0	C3	C2	C1	C0	Die Bits C0 bis C3 werden als Zeiger auf eines der Farbbregister im Bereich von COLOR00 bis COLOR15 verwendet. Dies ist identisch mit der normalen Farbwahl.
0	1	B3	B2	B1	B0	Der Rot- und Grünwert des letzten (linken) Pixels bleibt erhalten. Der alte Blauwert wird durch den Wert von B0 bis B3 ersetzt.
1	0	R3	R2	R1	R0	Der Blau- und Grünwert des letzten Pixels bleibt erhalten. Der alte Rotwert wird durch R0 bis R3 ersetzt.
1	1	G3	G2	G1	G0	Der Blau- und Rotwert des letzten Pixels bleibt erhalten. Der alte Grünwert wird durch G0 bis G3 ersetzt.

Bei dem ersten Punkt in einer Zeile wird die Rahmenfarbe (COLOR00) als Farbe des vorangegangenen Punktes verwendet.

Der Dual-Playfield-Modus

2. Farbwahl im Dual-Playfield-Modus

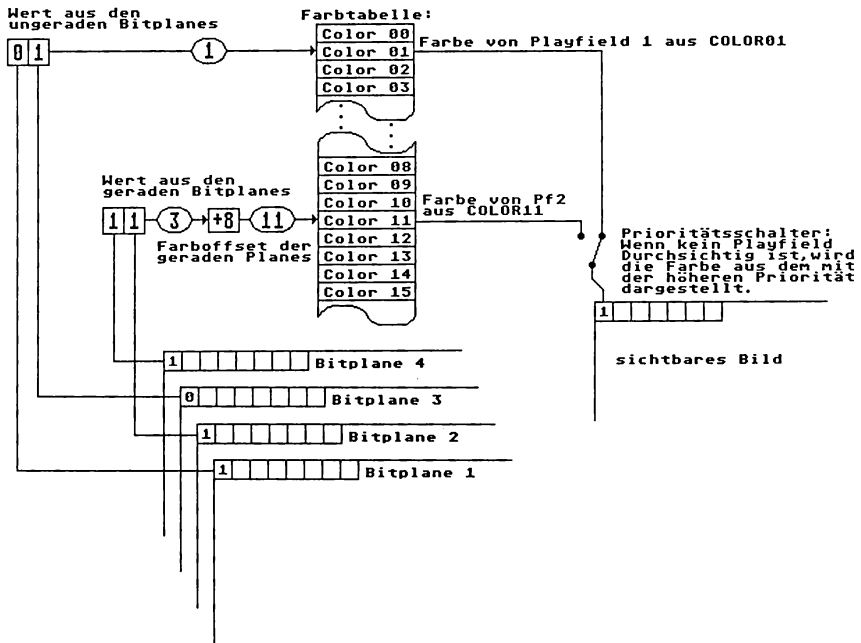


Abb. 1.5.5.2

Die bisher beschriebenen Modi verwendeten lediglich ein einziges Playfield. Der Dual-Playfield-Modus erlaubt die simultane Darstellung von zwei völlig voneinander unabhängigen Playfields. Es existieren dann quasi zwei Bildschirme, die auf einem

Monitor übereinander gelegt werden. Sie können (fast) völlig getrennt benutzt werden.

Dies ist besonders für Spiele interessant. Z.B. läßt sich damit sehr einfach ein Fernrohreffekt produzieren. Dazu wird das vordere Playfield mit schwarzen Punkten gefüllt. Lediglich in der Mitte bleibt ein Loch, durch das man dann einen Ausschnitt des hinteren Playfields sehen kann.

Jedes der beiden Playfields bekommt die Hälfte der aktivierten Bitplanes für seine Darstellung. Playfield 1 wird aus den ungeraden Planes (odd planes) gebildet, Playfield 2 besteht aus den geraden (even planes). Ist eine ungerade Anzahl von Bitplanes in Betrieb, hat Playfield 1 eine Bitplane mehr zur Verfügung.

Die Farbauswahl erfolgt im Dual-Playfield-Modus wie üblich: Der Wert, der aus den zu einem Punkt gehörigen Bits aller ungeraden (für Playfield 1) bzw. aller geraden (Playfield 2) Planes gebildet wird, dient als Zeiger auf einen Eintrag in der Farbtabelle. Da jedes Playfield aus höchstens 3 Planes bestehen kann, sind maximal acht Farben möglich. Diese werden bei Playfield 1 aus den unteren 8 Einträgen der Farbtabelle geholt (COLOR00 bis COLOR07). Bei Playfield 2 wird noch ein Offset von 8 zu dem Wert aus den Bitplanes dazuaddiert, wodurch sich seine Farben in den Positionen 8 bis 15 befinden (COLOR08 bis COLOR15).

Hat ein Punkt den Wert 0, wird seine Farbe nicht wie üblich aus COLOR00 (bzw. COLOR08) geholt, sondern er wird durchsichtig dargestellt. Dies bedeutet, daß dahinterliegende Bildelemente dort zu sehen sind. Dies können das andere Playfield, Sprites oder einfach der Hintergrund (COLOR00) sein.

Der Dual-Playfield-Modus ist auch in der hohen Auflösung anwendbar. Jedes Playfield hat dann nur noch vier Farben. An der Verteilung der Farbregister ändert sich nichts, allerdings sind die oberen vier Farbregister jedes Playfields unbenutzt (P1f1: COLOR04 bis 07, P1f2: COLOR12 bis 15).

Verteilung der Bitplanes im Dual-Playfield-Modus:

Bit-Planes	Planes in Playfield 1	Planes in Playfield 2
1	Plane 1	keine
2	Plane 1	Plane 2
3	Plane 1 und 3	Plane 2
4	Plane 1 und 3	Plane 2 und 4
5	Plane 1, 3 und 5	Plane 2 und 4
6	Plane 1, 3 und 5	Plane 2, 4 und 6

Farbauswahl im Dual-Playfield-Modus:

Playfield 1		Playfield 2	
Planes 5, 3, 1	Farbreg.	Planes 6, 4, 2	Farbreg.
0 0 0	Transparent	0 0 0	Transparent
0 0 1	COLOR01	0 0 1	COLOR09
0 1 0	COLOR02	0 1 0	COLOR10
0 1 1	COLOR03	0 1 1	COLOR11
1 0 0	COLOR04	1 0 0	COLOR12
1 0 1	COLOR05	1 0 1	COLOR13
1 1 0	COLOR06	1 1 0	COLOR14
1 1 1	COLOR07	1 1 1	COLOR15

Der Aufbau der Playfields

Wie schon erwähnt, besteht ein Playfield aus einer bestimmten Anzahl Bitplanes. Wie sehen diese Bitplanes aus? In 1.5.2 wurde schon beschrieben, daß sie als fortlaufender Speicherbereich konzipiert sind, wobei je nach Bildschirmbreite eine Zeile durch eine Anzahl von Worten repräsentiert wird. Im Normalfall sind dies 20 Worte in der niedrigen Auflösung (320 Punkte durch 16 Punkte pro Wort) und 40 (640/16) in der hohen.

Um den genauen Aufbau des Playfields festzulegen, sind folgende Schritte notwendig:

- Definition der gewünschten Bildgröße
- Setzen der Bitplane-Größe

- Anzahl der Bitplanes wählen
- Farbtabelle initialisieren
- Gewünschten Modus festlegen (Hires, Lores, HAM usw.)
- Copperlist aufbauen
- Copper initialisieren
- Copper- und Bitplane-DMA aktivieren

Festlegung der Bildgröße

Der Amiga erlaubt eine freie Positionierung der linken oberen und der rechten unteren Ecke des sichtbaren Bereichs des/der Playfields. Damit läßt sich sowohl die Bildposition als auch die Bildgröße variieren. Die Auflösung beträgt vertikal eine Rasterzeile und horizontal einen niedrig auflösenden Pixel. Zwei Register enthalten die Werte. DIWSTRT (Display Window Start) legt die horizontale und vertikale Startposition des Bildschirmfensters fest, d.h. die Zeile bzw. Spalte, in der die Darstellung des Playfields beginnt.

DIWSTOP (Display Window Stop) enthält die Endposition + 1. Damit ist die erste Zeile/Spalte nach dem Playfield gemeint. Soll es z.B. bis Zeile 250 gehen, muß man 251 als DIWSTOP-Wert angeben.

Außerhalb des sichtbaren Bereiches wird die Rahmenfarbe dargestellt. (Sie entspricht der Hintergrundfarbe und kommt aus dem COLOR00-Register.)

DIWSTRT \$08E (nur schreiben)

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

DIWSTOP \$90 (nur schreiben)

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
	V7	V6	V5	V4	V3	V2	V1	V0	H7	H6	H5	H4	H3	H2	H1	H0

Die in DIWSTRT festgelegte Startposition ist auf das obere linke Viertel des Bildschirms beschränkt, dies sind die Zeilen und Spalten 0 bis 255, da die fehlenden MSBs V8 und H8 als 0 angenommen werden. Gleiches gilt für die horizontale Endposition, nur wird hier H8 als 1 vorausgesetzt, womit sie innerhalb des Bereichs von 256 bis 458 liegt. Bei der vertikalen Endposition wurde ein anderer Weg begangen. Es sollten sowohl Positionen kleiner als auch größer 256 möglich sein. Aus diesem Grund wird das MSB der vertikalen Endposition, V8, durch Invertieren des V7-Bits erzeugt. Dadurch ist eine Endposition im Bereich der Zeilen von 128 bis 312 möglich. Bei Endpositionen von 256 bis 312 setzt man V7 auf 0 und damit V8 auf 1. Legt man V7 auf 1 geht V8 auf 0 und man erhält eine Position zwischen 128 und 255.

Das normale Bildschirmfenster hat eine obere linke Eckposition von horiz. 129 und vert. 41. Die untere rechte Ecke liegt bei 448, 296, d.h. DIWSTOP muß auf 449, 297 gesetzt werden. Die zugehörigen DIWSTRT- und DIWSTOP-Werte betragen hexadezimal \$2981 und \$29C1. Mit diesen Werten wird der normale Amiga-Screen von 640 auf 256 Punkten (bzw. 320 auf 256) in der Mitte des Bildschirms zentriert.

Warum verwendet man nicht den gesamten möglichen Bildschirmbereich? Dafür gibt es mehrere Gründe. Erstens stellt ein normaler Monitor nicht das gesamte Bild dar. Sein sichtbarer Bereich beginnt gewöhnlich erst einige Spalten bzw. Zeilen nach der jeweiligen Austastlücke. Außerdem ist eine Bildröhre nicht rechteckig. Würde man das Bildschirmfenster so hoch und breit wie die Monitorröhre einstellen, verdeckten die Ecken der Röhre ein Teil des Bildes.

Eine weitere Beschränkung erfahren die DIWSTRT- und DIWSTOP-Werte durch die Austastlücken. Vertikal fällt sie in den Bereich der Zeilen von 0 bis 25. Damit ist der sichtbare vertikale Bereich auf die Zeilen von 26 bis 312 (\$1A bis \$138) beschränkt. Innerhalb der Spalten 30 bis 106 (\$1E bis \$6A) liegt die horizontale Austastlücke. Es sind horizontale Positionen ab 107 (\$6B) möglich.

Nachdem man die Position des Bildschirmfensters festgelegt hat, müssen auch noch Anfang und Ende des Bitplane-DMA eingestellt werden. Damit die Punkte zum gewünschten Zeitpunkt auf dem Bildschirm erscheinen, müssen die Daten aus den Bitplanes rechtzeitig gelesen werden. Vertikal ist dies kein Problem. Der Bildschirm-DMA beginnt und endet in der selben Zeile wie das in DIWSTRT und DIWSTOP eingestellte Bildschirmfenster.

Horizontal ist es etwas komplizierter. Damit ein Punkt auf dem Bildschirm dargestellt werden kann, benötigt die Elektronik aus jeder Bitplane das aktuelle Wort. Bei 6 Bitplanes in der niedrigen Auflösung sind acht Buszyklen nötig, um alle Bitplanes zu lesen. In der hohen Auflösung sind es nur vier. (Zur Erinnerung: in einem Buszyklus werden 2 niedrig- oder 4 hochauflösende Punkte dargestellt.)

Zusätzlich braucht die Hardware noch einen halben Buszyklus, bevor die Daten auf dem Schirm erscheinen können. Der Bitplane-DMA muß also genau 8.5 Zyklen (17 Punkte) vor dem Anfang des Bildschirmfensters beginnen (4.5 Zyklen oder 9 Punkte in der hohen Auflösung, siehe Abbildung 1.5.2.3).

Der Buszyklus des ersten Bitplane-DMA in der Zeile wird in dem Register DDFSTRT (Display Data Fetch Start) abgelegt, der des letzten in DDFSTOP (Display Data Fetch Stop):

DDFSTRT \$092 (nur schreiben)

DDFSTOP \$094 (nur schreiben)

Bit Nr.:	15	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Funktion:	x	x	x	x	x	x	x	H8	H7	H6	H5	H4	H3	x	x

Die Auflösung beträgt acht Buszyklen im niedrigauflösenden Modus (Lores Modus), wobei H3 immer auf 0 liegt, und vier im hochauflösenden Modus (Hires Modus). Hier dient H3 als unterstes Bit. Der Grund für die beschränkte Auflösung liegt in der Aufteilung des Bitplane-DMA. Im Lores-Modus wird jede Bitplane alle acht Buszyklen einmal gelesen. Aus diesem Grund muß der DDFSTRT-Wert ein ganzzahliges Vielfaches von Acht

sein (H1 bis H3 = 0). Gleiches gilt für den Hires-Modus, nur sind es hier vier Buszyklen (H1 und H2 = 0). Allerdings sollte die Differenz aus DIWSTRT und DIWSTOP immer durch acht teilbar sein, da die Hardware, unabhängig von der Auflösung, die Zeile in Bereiche zu je 8 Buszyklen aufteilt. Auch im Hires-Modus wird der BitplaneDMA nach DIWSTOP noch 8 Buszyklen lang ausgeführt, dabei werden immer 32 Punkte gelesen.

Die korrekten Werte für DIWSTRT und DIWSTOP errechnen sich wie folgt:

Berechnung von DDFSTRT und DDFSTOP im Lores-Modus:

HStart = horizontaler Start des Bildschirmfensters

$DDFSTRT = (HStart/2 - 8.5) \text{ AND } \$FFF8$

$DDFSTOP = DDFSTRT + (PunkteproZeile/2 - 8)$

Dies ergibt für HStart = \$81 und 320 Punkte pro Zeile:

$DDFSTRT = (\$81/2 - 8.5) \text{ AND } \$FFF8 = \$38$

$DDFSTOP = \$38 + (320/2 - 8) = \$D0$

Berechnung von DDFSTRT und DDFSTOP im Hires-Modus:

$DDFSTRT = (HStart/2 - 4.5) \text{ AND } \$FFFC$

$DDFSTOP = DDFSTRT + (PunkteproZeile/4 - 8)$

Dies ergibt für HStart = \$81 und 640 Punkte pro Zeile:

$DDFSTRT = (\$81/2 - 4.5) \text{ AND } \$FFFC = \$3C$

$DDFSTOP = \$3C + (640/4 - 8) = \$D4$

DDFSTRT darf nicht kleiner als \$18 werden. DDFSTOP ist auf maximal \$D8 beschränkt. Die Gründe dafür kann man in Kapitel 1.5.2 nachlesen. Ein DDFSTRT-Wert kleiner \$28 hat keinen Sinn mehr, da die Punkte dann noch innerhalb der horizontalen Austastlücke dargestellt werden müßten, was nicht möglich ist (Ausnahme: Scrolling). Da sich bei DDFSTRT-Positionen kleiner \$34 die DMA-Zyklen der Bitplanes und Sprites überlappen, sind je nach DDFSTRT einige Sprites nicht mehr darstellbar.

Verschieben des Bildschirmfensters

Will man das Bildschirmfenster mittels DIWSTRT und STOP horizontal verschieben, kann es passieren, daß die Differenz zwischen DIWSTRT und DDFSTRT nicht genau 8.5 Buszyklen (17 Punkte) beträgt, da man DDFSTRT ja nur in Schritten von acht Buszyklen festlegen kann. In solch einem Fall würde ein Teil des ersten Datenwortes im unsichtbaren Bereich links neben der Grenze des Bildschirmfensters verschwinden. Um dies zu beheben, gibt es die Möglichkeit, die gelesenen Daten vor der Ausgabe auf dem Bildschirm soweit nach rechts zu verschieben, daß sie mit dem Beginn des Bildschirmfensters übereinstimmen. Wie man dies programmiert, wird im Abschnitt über das Scrolling genau erläutert.

Festlegen der Bitmap-Adressen

Die Werte in DDFSTRT und DDFSTOP bestimmen, wieviele Datenworte pro Zeile dargestellt werden. Für jede Bitmap muß jetzt die Startadresse gesetzt werde, damit der DMA-Controller weiß, von wo er die Punktdaten lesen soll. 12 Register enthalten diese Adressen. Je zwei Register, BPLxPTH und BPLxPTL, bilden sie gemeinsam für die Bitplane x. Zusammen einfach nur BPLxPT (BitPlanexPointer, Zeiger auf Bitplane x).

Adr.	Name	Funktion	
\$0E0	BPL1PTH	Anfangsadresse der	Bits 16-18
\$0E2	BPL1PTL	Bitplane 1	Bits 0-15
\$0E4	BPL2PTH	Anfangsadresse der	Bits 16-18
\$0E6	BPL2PTL	Bitplane 2	Bits 0-15
\$0E8	BPL3PTH	Anfangsadresse der	Bits 16-18
\$0EA	BPL3PTL	Bitplane 3	Bits 0-15
\$0EC	BPL4PTH	Anfangsadresse der	Bits 16-18
\$0F0	BPL4PTL	Bitplane 4	Bits 0-15
\$0F2	BPL5PTH	Anfangsadresse der	Bits 16-18
\$0F4	BPL5PTL	Bitplane 5	Bits 0-15
\$0F6	BPL6PTH	Anfangsadresse der	Bits 16-18
\$0F8	BPL6PTL	Bitplane 6	Bits 0-15

;unmögliche Bildschirmposition)

Das Zurücksetzen der BPLxPT ist unbedingt notwendig. Wenn man keine Copperlist verwenden will, muß man dies den Prozessor im Vertical-Blanking-Interrupt erledigen lassen.

Scrolling und extragroße Playfields

Die bisherigen Playfields hatten immer genau die Größe des Bildschirmfensters. Oft wäre es aber nützlich, ein großes Playfield im Speicher zu haben, von dem immer nur ein Ausschnitt im Bildschirmfenster zu sehen ist, den man aber ruckfrei in alle Richtungen verschieben kann. Dies ist beim Amiga problemlos möglich. Folgende Abschnitte zeigen dies sowohl in Richtung der X- als auch der Y-Achse.

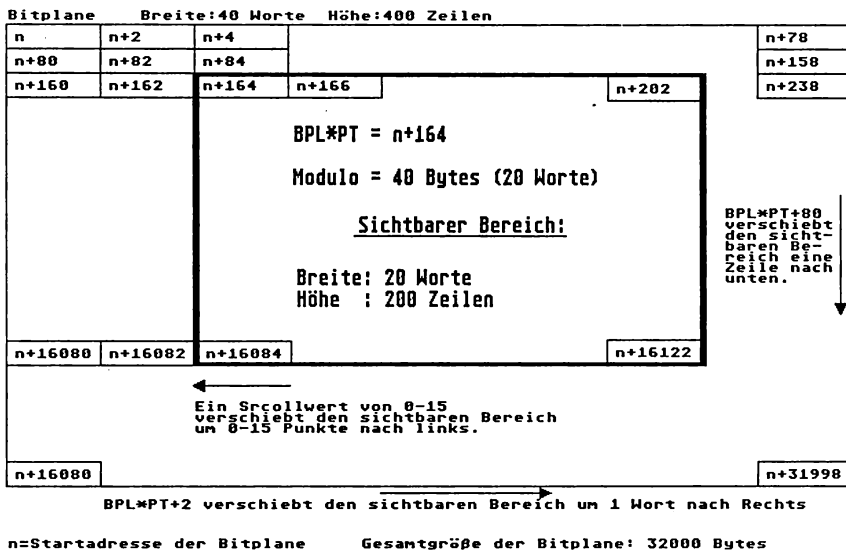


Abb. 1.5.5.3

Überhohe Playfields und vertikales Scrolling

Vertikal läßt sich dies sehr einfach realisieren. Man legt dazu wie üblich die notwendigen Bitplanes im Speicher an, aber diesmal mit mehr Zeilen als im Bildschirmfenster festgelegt. Z.B. bei einem 256 Zeilen hohen Standardfenster hätte ein doppelt so hohes Playfield im Speicher einfach 512 Zeilen. Um das Bildschirmfenster jetzt ruckfrei über dieses überhohe Playfield zu bewegen, verändert man die Werte der BPLxPT. Soll das Bildschirmfenster z.B. den Bereich von 100 bis 356 darstellen, muß der BPLxPT auf das erste Wort der 100. Zeile zeigen. Bei einer Bildbreite von 320 Punkten belegt jede Zeile 20 Worte (40 Byte) im Speicher. Multipliziert mit den 100 Zeilen ergibt sich eine Adresse von 4000. Plus der Anfangsadresse des großen Playfields erhält man so den gewünschten Wert, der in die BPLxPT muß. Um das Playfield im Bildschirmfenster zu scrollen, verändert man einfach diesen Wert mit jedem Bild um eine oder mehrere Zeilen, je nach gewünschter Scrollgeschwindigkeit. Da die BPLxPT nur außerhalb des sichtbaren Bereichs verändert werden dürfen, bedient man sich wieder obiger Copperlist. Man kann dann zu einem beliebigen Zeitpunkt die Adressen in der Copperlist verändern, der Copper schreibt sie automatisch zum richtigen Zeitpunkt in die BPLxPT-Register. Man muß lediglich aufpassen, daß man die Copperlist nicht verändert, während der Copper auf seine Befehle zugreift. Sonst kann es passieren, daß nachdem man erst ein Wort der Adresse geändert hat, der Copper plötzlich die gesamte, dann falsche Adresse liest.

Überbreite Playfields und horizontales Scrolling

Für das horizontale Scrolling und extrabreite Playfields sind spezielle Register vorhanden (ausschließlich Schreibregister):

\$108 BPL1MOD Modulo-Wert für die ungeraden Bitplanes

\$10A BPL2MOD Modulo-Wert für die geraden Bitplanes

BPLCON1 §102

Bit Nr:	15-8	7	6	5	4	3	2	1	0
Funktion:	unbel.	P2H3	P2H2	P2H1	P2H0	P1H3	P1H2	P1H1	P1H0

P1H0 - P1H3 Position der geraden Planes (vier Bits)

P2H0 - P2H3 Position der ungeraden Planes (vier Bits)

Die Modulo-Werte aus den BPLxMOD-Registern erlauben sogenannte rechteckige Speicherbereiche. Dieses Prinzip wird bei der Amiga-Hardware noch öfters verwendet. Es ermöglicht innerhalb eines großen Speicherbereichs, der in Zeilen und Spalten eingeteilt ist, einen kleineren zu definieren, der ebenfalls eine festgelegte Höhe und Breite besitzt. Nehmen wir an, der große Speicherbereich, in diesem Fall unser Playfield, wäre 640 Punkte breit und 256 hoch. Das ergäbe also 256 Zeilen zu je 40 Worten (80 Bytes). Der kleine entspricht dem Bildschirmfenster, es soll die normale Größe von 320 auf 200 Punkten haben, d.h. nur 20 Worte pro Zeile. Das Problem dabei ist, daß der BPLxPT bei der Ausgabe einer Zeile um 20 Worte erhöht wird. Um auf den Anfang der nächsten Zeile unseres Playfields zeigen zu können, müßte er aber 40 Worte weiter sein. Es müßte also nach jeder Zeile ein Wert von 20 Worten zu dem BPLxPT addiert werden. Genau dies kann der Amiga automatisch erledigen. Man schreibt die Differenz der beiden unterschiedlichen Zeilenlängen in die Modulo-Register. Nach der Ausgabe einer Zeile wird dieser Wert automatisch zu dem BPLxPT addiert.

Breite des Playfields: 80 Bytes (40 Worte)

Breite des Bildschirmfensters: 40 Bytes (20 Worte)

Notwendiger Modulo-Wert: 40 Bytes (Der Modulo-Wert muß immer in einer geraden Anzahl Bytes angegeben werden)

Start = Anfangsadresse der ersten Zeile des Playfields

Ausgabe der 1. Zeile:

Wort:.	0	1	2	3	19
BPLxPT:	Start	Start+2	Start+4	Start+6	... Start+38

Nach der Ausgabe des letztem Worts wird BPLxPT noch um 1 Wort erhöht: $\text{BPLxPT} = \text{Start} + 40$

Nach dem Zeilenende wird der Modulo-Wert zu BPLxPT addiert:

$$\text{BPLxPT} = \text{BPLxPT} + \text{Modulo BPLxPT} = \text{Start} + 40 + 40 = \text{Start} + 80$$

Ausgabe der 2. Zeile:

Wort:	0	1	2	3	...	19
BPLxPT:	Start+80	Start+82	Start+84	Start+86	...	Start+118

usw. Oben wurde die linke Hälfte der großen Bitmap im Bildschirmfenster dargestellt. Will man bei einer anderen horizontalen Position beginnen, addiert man einfach zu dem Anfangswert von BPLxPT die gewünschte Anzahl von Worten, wobei der Modulo-Wert gleich bleibt.

Die Startwerte sind wie oben. Lediglich BPLxPT steht nicht auf Start, sondern auf Start+40, damit wird die rechte Hälfte des großen Playfields angezeigt.

Ausgabe der 1. Zeile:

Wort:	0	1	2	3	...	19
BPLxPT:	Start+40	Start+42	Start+44	Start+46	...	Start+78

Nach Ausgabe des letzten Worts:

$$\text{BPLxPT} = \text{Start} + 80$$

Jetzt wird der Modulo-Wert zu BPLxPT addiert:

$$\text{BPLxPT} = \text{BPLxPT} + \text{Modulo BPLxPT} = \text{Start} + 80 + 40 = \text{Start} + 120$$

Ausgabe der 2. Zeile:

Wort:	0	1	2	3	...	19
BPLxPT:	Start+120	Start+122	Start+124	Start+126	...	Start+158

usw. Der Modulo-Wert läßt sich für die geraden und ungeraden Bitplanes getrennt einstellen. Dies ermöglicht im Dual-Playfield-Modus zwei unterschiedlich große Playfields. Arbeitet man nicht mit diesem Modus, setzt man beide BPLxMOD-Register auf den gleichen Modulo-Wert.

Mit Hilfe der BPLxPT- und der BPLxMOD-Register läßt sich der Bildschirm horizontal in Schritten von 16 Punkten verschieben. Ein feines Scrolling in Schritten von einem Punkt wird durch das BPLCON1-Register möglich. Die unteren vier Bits enthalten den Scrollwert der geraden Planes, Bits 4 bis 7 den der ungeraden. Dieser Scrollwert verzögert die Ausgabe der gelesenen Punktdaten für die entsprechenden Planes. Ist er auf Null, werden die Daten genau 8.5 (Hires: 4.5) Buszyklen nach der DDFSTRT-Position ausgegeben, ansonsten erscheinen sie bis zu fünfzehn Punkte später, je nach dem Scrollwert, d.h. das Bild verschiebt sich also innerhalb des feststehenden Bildschirmfensters um den Wert von BPLCON1 nach rechts.

Ein fließendes Scrolling des Bildschirminhalts nach rechts kann man erreichen, wenn man den Wert von BPLCON1 schrittweise von 0 auf 15 erhöht und dann wieder auf 0 zurücksetzt, wobei man gleichzeitig den BPLxPT wie oben beschrieben um 1 Wort erniedrigt.

Ein Scrolling nach links ergibt sich, wenn man den Scrollwert von 15 auf 0 herabzählt und dann BPLxPT um 1 Wort erhöht. BPLCON1 sollte nur außerhalb des sichtbaren Bereichs verändert werden. Entweder erledigt man dies innerhalb des Vertical-Blanking-Interrupts, oder man verwendet wieder den Copper. Dann kann man den Wert in der Copperlist zu jedem beliebigen Zeitpunkt ändern, er wird immer während der vertikalen Austastlücke in das BPLCON1-Register geschrieben.

Verschiebt man aber das Bild mittels des BPLCON1-Werts nach rechts, werden die überschüssigen Punkte zwar an der linken Kante korrekt abgeschnitten, an der rechten erscheinen aber keine neuen Punkte, da dort ja noch gar keine Punktdaten gelesen wurden. Um dies zu verhindern, muß der DDFSTRT-Wert gegenüber seinem normalen Beginn um 8 Buszyklen (Hires: 4

Buszyklen) vorverlegt werden. Man errechnet wie üblich anhand des gewünschten Bildschirmfensters den DDFSTRT-Wert und verringert ihn zusätzlich um 8 (bzw. 4). Aus dem normalen DDFSTRT-Wert von \$38 wird dann \$30 (Dadurch wird Sprite 7 abgeschaltet). Dieses zusätzlich gelesene Wort ist im Normalfall nicht sichtbar. Erst wenn der Scrollwert ungleich 0 ist, erscheinen seine Punkte in den freiwerdenden Positionen an der linken Kante des Bildschirmfensters. Hat dieses die Breite von 320 Punkten, werden jetzt 21 statt der üblichen 20 Datenworte pro Zeile gelesen. Bei der Berechnung der Bitplanes und der Modulo-Werte muß man das berücksichtigen.

Mit Hilfe des Scrollwerts kann man auch das Bildschirmfenster horizontal beliebig positionieren. Beträgt die Differenz aus DIWSTRT und DDFSTRT mehr als 17 Punkte, verschiebt man die gelesenen Daten einfach um eben diesen Mehrbetrag nach rechts.

Der Interlace-Modus

Obwohl im Interlace-Modus die doppelte Zeilenzahl darstellbar ist, unterscheidet er sich programmtechnisch nur durch einen geänderten Modulo-Wert und eine neue Copperlist von der normalen Darstellung. Wie im Kapitel 1.5.2 beschrieben, werden im Interlace-Modus mit jedem Bild abwechselnd die geraden oder die ungeraden Zeilen ausgegeben. Damit man ein Interlace-Playfield im Speicher normal darstellen kann, setzt man den Modulo-Wert gleich der Anzahl der Worte pro Zeile. Nach der Ausgabe einer Zeile wird dadurch noch einmal die Länge einer Zeile zum BPLxPT addiert, was gleichbedeutend mit dem Überspringen der nächsten Zeile ist. In jedem Bild wird nur jede zweite Zeile ausgegeben. Jetzt muß noch der BPLxPT in Übereinstimmung mit dem Halbbildtyp abwechselnd auf die erste oder die zweite Zeile des Playfields gesetzt werden, damit entweder die geraden oder die ungeraden Zeilen angezeigt werden. In einem Long Frame setzt man BPLxPT auf Zeile 1 (nur ungerade Zeilen), in einem Short Frame auf Zeile 2 (nur gerade Zeilen). Die Copperlist für ein Interlace-Playfield ist etwas komplizierter, es sind zwei Listen für die beiden Halbbildtypen nötig, die sich mit jedem Bild abwechseln:

Copperlist für ein Interlace-Playfield:

Zeile1 = Adresse der ersten Zeile der Bitplane

Zeile2 = Adresse der zweiten Zeile der Bitplane

Copper1:

```
MOVE #Zeile1Hi,BPLxPTH ;Zeiger für BPLxPT auf die Adresse
MOVE #Zeile1Lo,BPLxPTL ; der ersten Zeile setzen
... ;sonstige Copperbefehle
MOVE #Copper2Hi,COP1LCH ;Adresse der Copperlist auf Copper2
MOVE #Copper2Lo,COP1LCL ; setzen
WAIT ($FF,$FE) ;Ende der 1. Copperlist
```

Copper2:

```
MOVE #Zeile2Hi,BPLxPTH ;Zeiger für BPLxPT auf die Adresse
MOVE #Zeile2Lo,BPLxPTL ; der zweiten Zeile setzen
... ;sonstige Copperbefehle
MOVE #Copper1Hi,COP1LCH ;Adresse der Copperlist auf Copper1
MOVE #Copper1Lo,COP1LCL ; setzen
WAIT ($FF,$FE) ;Ende der 2. Copperlist
```

Der Copper wechselt nach jedem Bild selbständig seine Copperlist, indem er am Ende einer Befehlsliste die Adresse der anderen in COP1LC lädt. Diese Adresse wird mit dem Beginn des nächsten Bildes automatisch in den Programmzähler des Coppers geladen. Die Initialisierung des Interlace-Modus sollte mit Sorgfalt erfolgen, damit die Copperlist für die ungeraden Zeilen auch wirklich innerhalb eines Long Frames abgearbeitet wird:

- COP1LC auf Copper1 setzen
- LOF-Bit (Bit 15) im VPOS-Register (\$2A) auf 0 setzen. Dies stellt sicher, daß nach dem Einschalten des Interlace-Modus das erste Halbbild ein Long Frame ist und damit zu der Copperlist Copper1 paßt. Das LOF-Bit wird im Interlace-Modus mit jedem Halbbild invertiert. Setzt man es auf 0, springt es mit Beginn des nächsten Halbbildes auf 1. Dadurch ist dieses Halbbild dann sicher ein Long Frame.

- Interlace-Modus an
- Warten auf erste Zeile des nächsten Bildes (Zeile 0)
- Copper-DMA ein

Alle sonstigen Registerfunktionen bleiben im Interlace-Modus unverändert. Sämtliche Zeilenangaben (z.B. in DIWSTRT) beziehen sich immer auf die Zeilennummer innerhalb des aktuellen Halbbilds (0 - 311 für ein Short Frame und 0 - 312 für ein Long Frame). Schaltet man den Interlace-Modus ein, ohne andere Register zu verändern, bemerkt man lediglich ein leichtes Zittern des Bildes, da jetzt die Zeilen der Halbbilder gegeneinander versetzt werden, aber beide Bilder dieselben Grafikdaten enthalten. Erst wenn man mittels geeigneter Copperlist, doppelt so großen Bitplanes und entsprechenden Modulo-Werten dafür sorgt, das in jedem Halbbild andere Daten dargestellt werden, erreicht man die gewünschte Verdoppelung der Zeilenzahl.

Der Interlace-Modus bringt ein stärkeres Flimmern mit sich, da jede Zeile nur noch alle zwei Halbbilder und damit 25mal pro Sekunde aufgefrischt wird. Dieses Flimmern kann man auf ein Minimum herabsetzen, wenn zwischen den verschiedenen Farben in der Farbtabelle möglichst kleine Kontraste (geringe Helligkeitsunterschiede) bestehen. Denn das menschliche Auge kann bei niedrigem Kontrast schnelle Bildwechsel schlechter erkennen.

Die Kontrollregister

Um all die verschiedenen Modi zu aktivieren, existieren drei Kontrollregister, BPLCON0 bis BPLCON2. BPLCON1 enthält die Scrollwerte. Die beiden anderen sind wie folgt aufgebaut:

BPLCON0 \$100

Bit Nr.	Name	Funktion
15	HIRES	hochauflösender Modus ein (HIRES = 1)
14	BPU2	Die drei BPUx-Bits ergeben zusammen eine
13	BPU1	3-Bit-Zahl, die die Anzahl der verwendeten
12	BPU0	Bitplanes enthält (0 bis 6).
11	HOMOD	Hold-and-Modify ein (HOMOD = 1)
10	DBPLF	Dual-Playfield ein (DBPLF = 1)
9	COLOR	Videoausgang Farbe (COLOR = 1)
8	GAUD	Genlock Audio ein (GAUD = 1)
7-4	---	unbenutzt
3	LPEN	Lightpen-Eingang aktivieren (LPEN = 1)
2	LACE	Interlace-Modus ein (LACE = 1)
1	ERSY	Externe Synchronisation ein (ERSY = 1)
0	---	unbenutzt

HIRES

Mit dem Hires-Bit wird der hochauflösende Modus (Hires, 640 Punkte/Zeile) eingeschaltet.

BPL0 - BPL2

Diese drei Bits ergeben zusammen eine 3-Bit-Zahl, die die Anzahl der aktivierten Bitplanes auswählt. Es sind nur Werte zwischen 0 und 6 erlaubt.

HOMOD und DBPLF

Diese beiden Bits wählen den entsprechenden Modus aus. Sie dürfen nicht gemeinsam aktiv sein. Der Extra-Halfbright-Modus wird automatisch aktiviert, wenn man alle 6 Bitplanes einschaltet, aber weder HOMOD oder DBPLF anwählt.

LACE

Wenn das LACE-Bit gesetzt ist, wird das LOF-Frame Bit im VPOS-Register mit dem Anfang jedes neuen Bildes invertiert, wodurch der gewünschte Wechsel zwischen Long- und Short-frames stattfindet.

COLOR

Das Color-Bit schaltet den Colorburst-Ausgang von Agnus ein. Nur wenn Agnus dieses Colorburstsignal liefert, kann der Videomischer ein Farbvideosignal erzeugen. Ansonsten bleibt es Schwarz/Weiß. Der RGB-Ausgang wird davon nicht beeinflusst.

ERSY

Das ERSY-Bit schaltet die Anschlüsse für die vertikalen und horizontalen Synchronsignale von Ausgang auf Eingang um. Damit läßt sich das Amigabild durch externe Signale synchronisieren. Das Genlock-Interface nutzt dieses Bit, um das Amigabild mit einem beliebigem Videobild mischen zu können. Auch das GAUD-Bit ist für das Genlock-Interface gedacht (siehe Schnittstellen 1.3.2).

BPLCON2 §104

Bit Nr.:	15-7	6	5	4	3	2	1	0
Funktion:	unbel.	PF2PRI	PF2P2	PF2P1	PF2P0	PF1P2	PF1P1	PF1P0

PF2P0-PF2P2 und PF1P0-PF1P2 bestimmen die Priorität der Sprites in Bezug auf die Playfields (siehe nächstes Kapitel).

PF2PRI Ist dieses Bit gesetzt, haben die geraden Planes Priorität vor den ungeraden, d.h. sie erscheinen vor den ungeraden Planes. Das Bit hat nur im Dual-Playfield-Modus sichtbare Auswirkungen.

Aktivierung der Bildschirmdarstellung

Nachdem alle oben beschriebenen Register mit den gewünschten Werten versehen wurden, muß noch der DMA-Kanal für die Bitplanes eingeschaltet werden und, falls der Copper verwendet wird (was ja gewöhnlich der Fall ist), auch dessen DMA-Kanal. Folgender Move-Befehl erledigt dies, indem er die DMAEN-, BPLEN- und COPEN-Bits im DMA-Kontrollregister DMACON setzt:

```
MOVE.W #$8310,$DFF096
```

Beispielprogramme

Programm 1: Extra Half Bright Demo

Dieses Programm erzeugt ein Playfield mit den Standardmaßen 320 auf 256 Punkten im Lowres-Modus. Dabei werden 6 Bitplanes verwendet, wodurch automatisch der Extra-Halfbright-Modus aktiviert wird. Am Anfang belegt das Programm den notwendigen Speicher. Da erst jetzt die Adressen der einzelnen Bitplanes bekannt sind, wird die Copperlist nicht aus dem Programm herauskopiert, sondern direkt im Chip-RAM erstellt. Sie enthält nur die Befehle zum Setzen der BPLxPT-Register.

Damit man auch etwas von den 64 möglichen Farben sieht, zeichnet das Programm an zufälligen Positionen 16*16 Punkte große Blöcke in allen Farben. Als Zufallszahlengenerator wird dabei das VHPOS-Register verwendet.

```
;*** Demo für den Extra-Halfbright-Modus ***
```

```
;CustomChip-Register
```

```
INTENA = $9A ;Interrupt-Enable-Register (schreiben)
DMACON = $96 ;DMA-Kontrollregister (schreiben)
COLOR00 = $180 ;Farbpalettenregister 0
VHPOSR = $6 ;Strahlposition (lesen)
```


;Copper Register

COP1LC = \$80 ;Adresse der 1. Copperlist

COP2LC = \$84 ;Adresse der 2. Copperlist

COPJMP1 = \$88 ;Sprung nach Copperlist 1

COPJMP2 = \$8a ;Sprung nach Copperlist 2

;Bitplane Register

BPLCON0 = \$100 ;Bitplane Kontrollregister 0

BPLCON1 = \$102 ;1 (Scrollwerte)

BPLCON2 = \$104 ;2 (Sprite<>Playfield Priorität)

BPL1PTH = \$0E0 ;Zeiger auf 1. Bitplane

BPL1PTL = \$0E2 ;

BPL1MOD = \$108 ;Modulo-Wert für ungerade Bit-Planes

BPL2MOD = \$10A ;Module-Wert für gerade Bit-Planes

DIWSTRT = \$08E ;Start des Bildschirmfensters

DIWSTOP = \$090 ;Ende des Bildschirmfensters

DDFSTRT = \$092 ;Bit-Plane DMA Start

DDFSTOP = \$094 ;Bit-Plane DMA Stop

;CIA-A Portregister A (Maustaste)

CIAAPRA = \$bfe001

;Exec Library Base Offsets

OpenLibrary = -30-522 ;LibName,Version/a1,d0

Forbid = -30-102

Permit = -30-108

AllocMem = -30-168 ;ByteSize,Requirements/d0,d1

FreeMem = -30-180 ;MemoryBlock,ByteSize/a1,d0

;graphics base

StartList = 38

;sonstige Labels

Execbase = 4

Planesize = 40*256 ;Größe der Bitplane: 40 Bytes auf 256 Zeilen

CLsize = 13*4 ;Die Copperlist enthält 13 Befehle

Chip = 2 ;Chip-RAM anfordern

Clear = Chip+\$10000 ;Chip-RAM vorher löschen

```
;*** Vorprogramm ***
```

```
Start:
```

```
;Speicher für die Bitplanes anfordern
```

```
move.l Execbase,a6
move.l #Planesize*6,d0 ;Speicherbedarf aller Planes
move.l #clear,d1 ;Speicher soll mit Nullen gefüllt werden
jsr AllocMem(a6) ;Speicher anfordern
move.l d0,Planeadr ;Adresse der ersten Plane speichern
beq Ende ;Fehler! -> Ende
```

```
;Speicher für Copperlist anfordern
```

```
moveq #Clsize,d0 ;Größe der Copperlist
moveq #chip,d1
jsr AllocMem(a6)
move.l d0,CLadr
beq FreePlane ;Fehler! -> RAM für Bitplanes wieder freigeben
```

```
;Copperlist erstellen
```

```
moveq #5,d4 ;6 Planes = 6 Schleifendurchläufe
move.l d0,a0 ;Adresse der Copperlist nach a0
move.l Planeadr,d1
move.w #bpl1pth,d3 ;erstes Register nach d3
```

```
MakeCL: move.w d3,(a0)+ ;BPLxPTH ins RAM
addq.w #2,d3 ;nächstes Register
swap d1
move.w d1,(a0)+ ;Hi-Wort der Planeadresse ins RAM
move.w d3,(a0)+ ;BPLxPTL ins RAM
addq.w #2,d3 ;nächstes Register
swap d1
move.w d1,(a0)+ ;Lo-Wort der Planeadresse ins RAM
add.l #planesize,d1 ;Adresse der nächsten Plane berechnen
```

```
dbf d4,MakeCL
```

```
move.l #$fffffffe,(a0) ;Ende der Copperlist
```

```
;*** Hauptprogramm ***

;DMA und Taskswitching sperren

jsr forbid(a6)
lea $dff000,a5
move.w #$03e0,dmacon(a5)

;Copper initialisieren

move.l CLadr,cop1lc(a5)
clr.w copjmp1(a5)

;Farbtabelle mit unterschiedlichen Farben füllen

moveq #31,d0 ;Zähler für Farbregister
lea color00(a5),a1
moveq #1,d1 ;erste Farbe
SetTab:
move.w d1,(a1)+ ;Farbe in Farbregister
mulu #3,d1 ;nächste Farbe berechnen
dbf d0,SetTab

;Playfield initialisieren

move.w #$3081,diwstrt(a5) ;Standardwerte für
move.w #$30c1,diwstop(a5) ;Bildschirmfenster
move.w #$0038,ddfstrt(a5) ;und BitplaneDMA
move.w #$00d0,ddfstop(a5)
move.w #%0110001000000000,bplcon0(a5) ;6 Bitplanes
clr.w bplcon1(a5) ;kein Scrolling
clr.w bplcon2(a5) ;Priorität ist egal
clr.w bpl1mod(a5) ;Modulo für alle Planes gleich Null
clr.w bpl2mod(a5)

;DMA ein
move.w #$8380,dmacon(a5)

;Bitplane modifizieren

moveq #40,d5 ;Bytes pro Zeile
clr.l d2 ;Mit Farbe 0 beginnen

Loop: clr.l d0
move.w vposr(a5),d0 ;Zufallswert nach d0
and.w #$3ffe,d0 ;überflüssige Bits ausmaskieren
```

```

cmp.w #$2580,d0 ;Größer als Plane?
bcs Weiter ;Wenn nein, dann weiter
and.w #$1ffe,d0 ;sonst oberes Bit löschen
Weiter: move.l Planeadr,a4 ;Adresse der 1.Bitplane nach a4
add.l d0,a4 ;Adresse des Blocks berechnen
moveq #5,d4 ;Zähler für Bitplanes
move.l d2,d3 ;Farbe in Arbeitsregister

```

Block:

```

clr.l d1
lsr #1,d3 ;ein Bit aus Farbnummer in X-Flag
negx.w d1 ;d1 an X-Flag angleichen
moveq #15,d0 ;16 Zeilen pro Block
move.l a4,a3 ;Blockadresse in Arbeitsregister

```

Fill:

```

move.w d1,(a3) ;Wort in Bitplane
add.l d5,a3 ;nächste Zeile errechnen
dbf d0,Fill

```

```

add.l #Planesize,a4 ;nächste Bitplane
dbf d4,Block

```

```

addq.b #1,d2 ;nächste Farbe
btst #6,ciaapra ;Maustaste gedrückt?
bne Loop ;Nein -> weitermachen

```

;*** Nachprogramm ***

;Alte Copperlist wieder aktivieren

```

move.l #GRname,a1 ;Parameter für OpenLibrary setzen
clr.l d0
jsr OpenLibrary(a6) ;Graphics Library öffnen
ove.l d0,a4
move.l StartList(a4),cop1lc(a5) ;Adresse der Startlist
clr.w copjmp1(a5)
move.w #$8060,dmacon(a5) ;abgeschaltete DMA-Kanäle wieder an
jsr permit(a6) ;Task-Switching wieder erlauben

```

;Speicher für Copperlist wieder freigeben

```

move.l CLadr,a1 ;Parameter für FreeMem setzen
moveq #CLsize,d0
jsr FreeMem(a6) ;Speicher freigeben

```

```
;Speicher für Bitplanes wieder freigeben
FreePlane:
move.l Planeadr,a1
move.l #Planesize*6,d0
jsr FreeMem(a6)
```

```
Ende:
clr.l d0
rts ;Programm verlassen
```

```
;Variablen
```

```
CLadr: dc.l 0
Planeadr: dc.l 0
```

```
;Konstanten
```

```
GRname: dc.b "graphics.library",0
```

```
;Programmende
```

Programm 2: Dual-Playfield & Smooth Scrolling

Dieses Programm verwendet mehrere Effekte auf einmal: Erstens schafft es einen Dual-Playfield-Bildschirm mit einer Lowres-Bitplane pro Playfield. Dann vergrößert es das normale Bildschirmfenster so, daß keine Ränder mehr zu sehen sind, und zuletzt scrollt es Playfield 1 horizontal und Playfield 2 vertikal.

Am Anfang und Ende werden wieder die üblichen Routinen zur Speicherbelegung usw. verwendet.

Beide Playfields werden mit einem Schachbrettmuster aus 16*16 Punkten großen Feldern gefüllt.

Die Hauptschleife des Programms, die das Scrolling ausführt, wartet zuerst auf eine Zeile innerhalb der vertikalen Austastlücke, in der das Betriebssystem schon alle eventuellen Interruptroutinen abgearbeitet und der Copper die BPLxPT gesetzt hat. Danach zählt es den vertikalen Scroll-Zähler hoch, errechnet den neuen BPLxPT für Playfield 2 und schreibt ihn in die Copperlist.

Die horizontale Scroll-Position entsteht durch Trennen des Scroll-Zählers in die unteren 4 Bits und den Rest. Die unteren 4 Bits werden in den Scroll-Wert für Playfield 1 im BPLCON1-Register geschrieben, aus dem 5. Bit wird der neue BPLxPT errechnet und in die Copperlist kopiert.

Sowohl der horizontale als auch der vertikale Scroll-Zähler werden von 0 bis 31 hochgezählt und dann wieder auf 0 zurückgesetzt. Dies reicht für den Effekt eines fließenden Scrollings, da sich der Inhalt der Playfields aufgrund des verwendeten Musters alle 32 Punkte wiederholt.

*** Dual-Playfield & Scroll Demo ***

;CustomChip-Register

INTENA = \$9A ;Interrupt-Enable-Register (schreiben)
 INTREQR = \$1e ;Interrupt-Request-Register (lesen)
 DMACON = \$96 ;DMA-Kontrollregister (schreiben)
 COLOR00 = \$180 ;Farbpalettenregister 0
 VPOSR = \$4 ;Strahlposition (lesen)

;Copper Register

COP1LC = \$80 ;Adresse der 1. Copperlist
 COP2LC = \$84 ;Adresse der 2. Copperlist
 COPJMP1 = \$88 ;Sprung nach Copperlist 1
 COPJMP2 = \$8a ;Sprung nach Copperlist 2

;Bitplane Register

BPLCON0 = \$100 ;Bitplane Kontrollregister 0
 BPLCON1 = \$102 ;1 (Scrollwerte)
 BPLCON2 = \$104 ;2 (Sprite<>Playfield Priorität)
 BPL1PTH = \$0E0 ;Zeiger auf 1. Bitplane
 BPL1PTL = \$0E2 ;
 BPL1MOD = \$108 ;Modulo-Wert für ungerade Bit-Planes
 BPL2MOD = \$10A ;Module-Wert für gerade Bit-Planes
 DIWSTRT = \$08E ;Start des Bildschirmfensters
 DIWSTOP = \$090 ;Ende des Bildschirmfensters
 DDFSTRT = \$092 ;Bit-Plane DMA Start
 DDFSTOP = \$094 ;Bit-Plane DMA Stop

CIA-A Portregister A (Maustaste)

CIAAPRA = \$bfe001

;Exec Library Base Offsets

OpenLibrary = -30-522 ;LibName,Version/a1,d0

Forbid = -30-102

Permit = -30-108

AllocMem = -30-168 ;ByteSize,Requirements/d0,d1

FreeMem = -30-180 ;MemoryBlock,ByteSize/a1,d0

;graphics base

StartList = 38

;sonstige Labels

Execbase = 4

Planesize = 52*345 ;Größe der Bitplane

Planewidth = 52

CLsize = 5*4 ;Die Copperlist enthält 5 Befehle

Chip = 2 ;Chip-RAM anfordern

Clear = Chip+\$10000 ;Chip-RAM vorher löschen

;*** Vorprogramm ***

Start:

;Speicher für die Bitplanes anfordern

move.l Execbase,a6

move.l #Planesize*2,d0 ;Speicherbedarf der Planes

move.l #clear,d1

jsr AllocMem(a6) ;Speicher anfordern

move.l d0,Planeadr

beq.L Ende ;Fehler! -> Ende

;Speicher für Copperlist anfordern

moveq #CLsize,d0

moveq #chip,d1

jsr AllocMem(a6)

move.l d0,CLadr

beq.L FreePlane ;Fehler! -> Speicher der Planes freigeben

```
;Copperlist erstellen
```

```
moveq #1,d4 ;zwei Bitplanes
move.l d0,a0
move.l Planeadr,d1
move.w #bpl1pth,d3
```

```
MakeCL: move.w d3,(a0)+
addq.w #2,d3
swap d1
move.w d1,(a0)+
move.w d3,(a0)+
addq.w #2,d3
swap d1
move.w d1,(a0)+
add.l #planesize,d1 ;Adresse der nächsten Plane
```

```
dbf d4,MakeCL
```

```
move.l #$ffffffe,(a0) ;Ende der Copperlist
```

```
;*** Hauptprogramm ***
```

```
;DMA und Taskswitching sperren
```

```
jsr forbid(a6)
lea $dff000,a5
move.w #$01e0,dmacon(a5)
```

```
;Copper initialisieren
```

```
move.l CLadr,cop1lc(a5)
clr.w copjmp1(a5)
```

```
;Playfield initialisieren
```

```
move.w #0,color00(a5)
move.w #$0f00,color00+2(a5)
move.w #$000f,color00+18(a5)
move.w #$1a64,diwstrt(a5) ;26,100
move.w #$39d1,diwstop(a5) ;313,465
move.w #$0020,ddfstrt(a5) ;ein zusätzliches Wort lesen
move.w #$00d8,ddfstop(a5)
move.w #%0010011000000000,bplcon0(a5) ;Dual-Playfield an
clr.w bplcon1(a5) ;Scrollwert am Anfang auf 0
clr.w bplcon2(a5) ;Playfield 1 vor Playfield 2
```



```
move.w #4,bpl1mod(a5) ;Modulo auf 2 Worte
move.w #4,bpl2mod(a5)

;DMA ein
move.w #$8180,dmacon(a5)

;Bitplanes mit Schachbrettmuster füllen

move.l planeadr,a0
move.w #planesize/2-1,d0 ;Schleifenzähler
move.w #13*16,d1 ;Höhe = 16 Zeilen
move.l #$ffff0000,d2 ;Schachbrettmuster
move.w d1,d3

fill: move.l d2,(a0)+
      subq.w #1,d3
      bne.s weiter
      swap d2 ;Muster wechseln
      move.w d1,d3
      weiter: dbf d0,fill

;Playfields scrollen

clr.l d0 ;vertikale Scrollposition
clr.l d1 ;horizontale Scrollposition
move.l CLadr,a1 ;Adresse der Copperlist
move.l Planeadr,a0 ;Adresse der ersten Bitplane

;Auf Rasterzeile 16 warten (nach den Exec-Interrupts)

wait: move.l vposr(a5),d2 ;Position lesen
and.l #$0001FF00,d2 ;horizontale Bits ausmaskieren
cmp.l #$00001000,d2 ;auf Zeile 16 warten
bne.s wait

;Playfield 1 vertikal scrollen

addq.b #2,d0 ;vertikalen Scrollzähler erhöhen
cmp.w #$80,d0 ;schon auf 128 (4*32)?
bne.s novover
clr.l d0 ;Dann zurück auf 0
novover:
move.l d0,d2 ;Scrollzähler kopieren
lsr.w #2,d2 ;Kopie durch 4 dividieren
mulu #52,d2 ;Anzahl Bytes pro Zeile * Scrollposition
add.l a0,d2 ;plus Adresse erster Plane
```

```
add.l #Planesize,d2 ;plus Planegröße
move.w d2,14(a1) ;ergibt Endadresse für Copperlist
swap d2
move.w d2,10(a1)
```

```
;Playfield 2 horizontal scrollen
```

```
addq.b #1,d1 ;horizontalen Scrollzähler erhöhen
cmp.w #$80,d1 ;schon auf 128 (4*32)
bne.s nohover
clr.l d1 ;Dann zurück auf 0
nohover:
move.l d1,d2 ;Scrollzähler kopieren
lsr.w #2,d2 ;Kopie durch 4 dividieren
move.l d2,d3 ;Scrollposition kopieren
and.w #$FFF0,d2 ;untere 4 Bit ausmaskieren
sub.w d2,d3 ;untere 4 Bit in d3 isolieren
move.w d4,bplcon1(a5) ;letzten Wert in BPLCON1
move.w d3,d4 ;neuen Scrollwert nach d4
lsr.w #3,d2 ;neue Adresse für Copperlist
add.l a0,d2 ;ausrechnen
move.w d2,6(a1) ;und in Copperlist schreiben
swap d2
move.w d2,2(a1)
```

```
btst #6,ciaapra ;Maustaste gedrückt?
bne.s wait ;Nein -> weitermachen
```

```
;*** Nachprogramm ***
```

```
;Alte Copperlist wieder aktivieren
```

```
end: move.l #GRname,a1 ;Parameter für OpenLibrary setzen
clr.l d0
jsr OpenLibrary(a6) ;Graphics Library öffnen
move.l d0,a4
move.l StartList(a4),cop1lc(a5)
clr.w copjmp1(a5)
move.w #$83e0,dmacon(a5)
jsr permit(a6) ;Task-Switching einschalten
```

```
;Speicher für Copperlist wieder freigeben
```

```
move.l CLadr,a1 ;Parameter für FreeMem setzen
moveq #CLsize,d0
jsr FreeMem(a6) ;Speicher freigeben
```

;Speicher für Bitplanes wieder freigeben

FreePlane:

move.l Planeadr,a1

move.l #Planesize*2,d0

jsr FreeMem(a6)

Ende:

clr.l d0

rts;Programm verlassen

;Variablen

CLadr: dc.l 0

Planeadr: dc.l 0

test: dc.l 0

;Konstanten

GRname: dc.b "graphics.library",0

;Programmende

1.5.6 Sprites

Sprites sind kleine Grafik-Elemente, die völlig unabhängig von den Playfields verwendet werden können. Jedes Sprite ist 16 Punkte breit und kann maximal die Höhe des ganzen Bildschirmfensters haben. Es kann eine beliebige Position auf dem Bildschirm einnehmen. Normalerweise befindet sich ein Sprite vor dem/den Playfield(s). Seine Punkte verdecken die der dahinterliegenden Grafik. Der Mauszeiger wird z.B. durch ein Sprite dargestellt. Beim Amiga sind maximal acht Sprites möglich. Ein Sprite ist normalerweise dreifarbig, es besteht aber die Möglichkeit, zwei Sprites zu einem neuen Sprite mit fünfzehn Farben zu kombinieren.

Der Aufbau der Sprites

Farbwahl

Die Farbwahl bei den Sprites ähnelt sehr derjenigen eines Dual-Playfield-Bildschirms. Ein Sprite ist sechzehn Punkte breit, die von zwei Datenworten repräsentiert werden, die sozusagen zwei "Mini-Bitplanes" darstellen. Denn wie bei den Bitplanes wird die Farbe eines Punktes von zusammengehörigen Bits aller Bitplanes gebildet. Bei einem Sprite wird die Farbe des ersten Punktes (dies ist der am weitesten links stehende Punkt des Sprites) durch die beiden höchstwertigen Bits (Bit 15) beider Datenworte ausgewählt. Die beiden niederwertigsten Bits (Bit 0) bestimmen die Farbe des letzten Punktes. Jeder Punkt wird also von einem 2 Bit Datenwert repräsentiert, der vier verschiedene Werte annehmen kann. Um aus diesem Wert die tatsächliche Farbe des Punktes zu gewinnen, wird wieder die Farbtabelle verwendet. Es gibt keine eigenen Farbregister für Sprites. Allerdings werden die Sprite-Farben aus der hinteren Tabellenhälfte geholt, also aus den Farbregistern 16 - 31. Damit kommen Sprite- und Playfield-Farben erst miteinander in Konflikt, wenn Playfields mit mehr als 16 Farben kreiert werden.

Folgende Tabelle zeigt die Zugehörigkeit von Farbregistern und Sprites:

Sprite Nr.	Sprite-Daten	Farbregister
0 & 1	0 0	durchsichtig
	0 1	COLOR17
	1 0	COLOR18
	1 1	COLOR19
2 & 3	0 0	durchsichtig
	0 1	COLOR21
	1 0	COLOR22
	1 1	COLOR23
4 & 5	0 0	durchsichtig
	0 1	COLOR25
	1 0	COLOR26
	1 1	COLOR27

6 & 7	0 0	durchsichtig
	0 1	COLOR29
	1 0	COLOR30
	1 0	COLOR31

Je zwei aufeinanderfolgende Sprites haben dieselben Farbregister.

Wie schon im Dual-Playfield-Modus wird die Bitkombination von zwei Nullen nicht in einer bestimmten Farbe dargestellt, sondern erscheint durchsichtig. An diesen Stellen sieht man die Farbe des dahinterliegenden Bildelements durch, egal, ob es sich nun um ein anderes Sprite, ein Playfield oder nur den Hintergrund handelt.

Reichen einem die drei Farben nicht aus, kann man zwei Sprites miteinander kombinieren. Die zwei Bit-Kombinationen beider Sprites ergeben zusammen einen 4-Bit-Wert. Es kann immer nur ein Sprite mit einer geraden Nummer mit dem darauffolgenden ungeraden Sprite kombiniert werden. Also Nr. 0 mit Nr. 1, 2 mit 3 usw. Dabei ergeben die beiden Datenworte aus dem Sprite mit der höheren Nummer auch die beiden höchstwertigen Bits des gemeinsamen 4-Bit-Werts. Dieser dient dann als Zeiger auf eines von fünfzehn Farbregistern, wobei der Wert Null wieder transparent erscheint. Die Farbregister sind für alle vier möglichen Spritekombinationen dieselben: COLOR16 bis COLOR31.

Sprite-Daten	Farbregister	Sprite-Daten	Farbregister
0 0 0 0	durchsichtig	1 0 0 0	COLOR24
0 0 0 1	COLOR17	1 0 0 1	COLOR25
0 0 1 0	COLOR18	1 0 1 0	COLOR26
0 0 1 1	COLOR19	1 0 1 1	COLOR27
0 1 0 0	COLOR20	1 1 0 0	COLOR28
0 1 0 1	COLOR21	1 1 0 1	COLOR29
0 1 1 0	COLOR22	1 1 1 0	COLOR30
0 1 1 1	COLOR23	1 1 1 1	COLOR31

Der Sprite-DMA

Die Programmierung der Sprites ist auf dem Amiga sehr komfortabel gelöst worden. Fast die gesamte Arbeit wird von den Sprite-DMA-Kanälen erledigt. Um ein Sprite auf dem Bildschirm darzustellen, benötigt man nur eine spezielle Sprite-Datenliste im Speicher. Sie enthält fast alle für das Sprite notwendigen Daten. Lediglich die Adresse dieser Liste muß man dem DMA-Controller noch mitteilen, um das Sprite erscheinen zu lassen.

Der DMA-Controller hat für jedes Sprite einen DMA-Kanal. Dieser kann in jeder Rasterzeile nur zwei Datenworte lesen. Aus diesem Grund ist ein normales Sprite auf eine Breite von 16 Punkten und vier Farben beschränkt. Da diese beiden Datenworte in jeder Zeile gelesen werden können, ist die Höhe eines Sprites nur von dem Bildschirmfenster begrenzt, nach dem sich auch die Sprites zu richten haben.

Der Aufbau einer Sprite-Datenliste

Eine solche Datenliste besteht aus einzelnen Zeilen, die jeweils zwei Datenworte enthalten. In jeder Rasterzeile wird eine dieser Zeilen per DMA gelesen. Sie kann entweder zwei Kontrollworte enthalten, die das Sprite initialisieren, oder zwei Datenworte mit den Punktdaten, wenn das Sprite gerade dargestellt wird.

Die Kontrollworte bestimmen die horizontale Spalte und die erste und letzte Zeile des Sprites.

Nachdem der DMA-Controller diese Worte gelesen und in den entsprechenden Registern plazierte hat, wartet er, bis der Elektronenstrahl die Anfangszeile des Sprites erreicht. Dann werden mit jeder Zeile zwei Datenworte gelesen und von der Hardware in Denise an der entsprechenden horizontalen Position auf dem Bildschirm ausgegeben, bis die letzte Zeile des Sprites vorbei ist. Die nächsten beiden Worte in der Sprite-Datenliste werden wieder als Kontrollworte aufgefaßt. Sind beide gleich 0, beendet der DMA-Kanal seine Aktivität. Es ist aber auch möglich, eine neue Sprite-Position anzugeben. Der DMA-Controller wartet

dann erneut auf die Startzeile, und der Vorgang wiederholt sich solange, bis zwei Kontrollworte mit dem Wert 0 als Endmarkierung der Liste gefunden werden.

Aufbau einer Sprite-Datenliste (Start = Anfangsadresse der Liste im Chip-RAM):

Adresse	Inhalt
Start+4	1. u. 2. Datenwort der 1. Zeile des Sprites
Start+8	1. u. 2. Datenwort der 2. Zeile des Sprites
Start+12	1. u. 2. Datenwort der 3. Zeile des Sprites
Start+4*n	1. u. 2. Datenwort der n. Zeile des Sprites
Start+4*(n+1)	0,0 Ende der Sprite-Datenliste

Aufbau des ersten Kontrollworts:

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Funktion:	E7	E6	E5	E4	E3	E2	E1	E0	H8	H7	H6	H5	H4	H3	H2	H1

Aufbau des zweiten Kontrollworts:

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Funktion:	L7	L6	L5	L4	L3	L2	L1	L0	AT	0	0	0	0	E8	L8	H0

H0 bis H8	Horizontale Position des Sprites (HSTART)
E0 bis E8	Erste Zeile des Sprites (VSTART)
L0 bis L8	Letzte Zeile des Sprites+1 (VSTOP)
AT	Attach-Kontrollbit

Es sind je 9 Bits für die horizontale und vertikale Position des Sprites vorgesehen. Diese Bits sind allerdings etwas unpraktisch über die beiden Kontrollregister verteilt.

Die Auflösung beträgt in horizontaler Richtung einen niedrig-auflösenden Punkt, in vertikaler Richtung eine Rasterzeile. Diese Werte sind unveränderlich, da unabhängig von dem eingestellten Modus des/der Playfields.

Die Sprites sind auf das Bildschirmfenster (eingestellt in DIWSTART und DIWSTOP) beschränkt. Liegen die in den Kontrollworten festgelegten Koordinaten außerhalb, werden die Sprites nur noch teilweise oder überhaupt nicht mehr dargestellt, da alle Punkte, die sich nicht innerhalb des Bildschirmfensters befinden, abgeschnitten werden.

Die horizontale und vertikale Startposition bezieht sich auf die linke, obere Ecke des Sprites. Die vertikale Stopposition legt die erste Zeile nach dem Sprite fest, d.h. die letzte Zeile des Sprites+1. Die Anzahl der Zeilen eines Sprites ist also VSTOP-VSTART.

Folgende Beispielliste stellt ein Sprite an den Koordinaten 180,160 dar, also etwa in Bildschirmmitte. Er soll eine Höhe von 8 Zeilen haben. Die letzte Zeile (VSTOP) ist also 168.

Faßt man die beiden Datenworte zusammen, erhält man Zahlen von 0 bis 3, die jeweils eine der drei Sprite-Farben oder die durchsichtigen Punkte repräsentieren. Dadurch kann man sich das Sprite besser vorstellen:

```
0000002222000000
0000220000220000
0002200330022000
0022003113002200
0022003113002200
0002200330022000
0000220000220000
0000002222000000
```

In der Datenliste muß man beide Worte getrennt angeben:

Start:

```
dc.w $A05A,$A800 ;HSTART = $B4, VSTART = $A0, VSTOP = $A8
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0011 1100 0000,%0011 0010 0100 1100
dc.w %0000 0001 1000 0000,%0001 1001 1001 1000
```



```
dc.w %0000 0000 0000 0000,%0000 1100 0011 0000
dc.w %0000 0000 0000 0000,%0000 0011 1100 0000
dc.w 0,0 ;Ende der Sprite-Datenliste
```

Das AT-Bit im 2. Kontrollwort legt fest, ob zwei Sprites miteinander kombiniert werden. Es hat nur in den Sprites mit ungeraden Nummern eine Funktion (Sprites 1, 3, 5, 7). Ist es z.B. für Sprite 1 gesetzt, werden dessen Datenbits mit denen von Sprite 0 gemeinsam als vier Bit-Zeiger auf die Farbtabelle interpretiert. Die Reihenfolge der Bits ist dabei folgendermaßen:

Sprite 1 (ungerade Nummer), zweites Datenwort:	Bit 3 (MSB)
Sprite 1, erstes Datenwort:	Bit 2
Sprite 0 (gerade Nummer), zweites Datenwort:	Bit 1
Sprite 0, erstes Datenwort:	Bit 0 (LSB)

Sollen zwei Sprites auf diese Weise miteinander kombiniert werden, muß auch ihre Position übereinstimmen. Ist dies nicht der Fall, wird automatisch auf die alte, dreifarbige Darstellung zurückgeschaltet. Am einfachsten ist es, in beide Sprite-Datenlisten dieselben Kontrollworte zu schreiben. Nun noch als Beispiel eine Sprite-Datenliste für ein fünfzehnfarbiges Sprite:

Der Einfachheit halber soll unser Sprite nur aus 4 Zeilen bestehen. Die Ziffern stellen wieder die Farbe der entsprechenden Punkte da. Um alle fünfzehn Farben plus transparent darstellen zu können, wurden die Hexadezimalziffern "A" bis "F" dazugenommen.

```
0011111111111100
1123456789ABCD11
11EFEFEFEFEFEF11
0011111111111100
```

Die Bildung der notwendigen Datenworte kann man gut aus Zeile 2 ansehen:

```

Farben des Sprite:      1123456789ABCD11
Sprite1, Datenwort2:   0000000011111100
Sprite1, Datenwort1:   0000111100001100
Sprite0, Datenwort2:   0011001100110000
Sprite0, Datenwort1:   1101010101010111

```

Die Datenlisten für das gesamte Sprite sehen folgendermaßen aus:

Horizontale Position (HSTART) sei wieder 180. Erste Zeile des Sprites (VSTART) 160, letzte Zeile (VSTOP) 164.

```

StartSprite0:
dc.w $A05A,$A400 ;HSTART=$B4, VSTART=$A0, VSTOP=$A4, AT=0
dc.w %0011 1111 1111 1100,%0000 0000 0000 0000
dc.w %1101 0101 0101 0111,%0011 0011 0011 0000
dc.w %1101 0101 0101 0111,%0011 1111 1111 1100
dc.w %0011 1111 1111 1100,%0000 0000 0000 0000
dc.w 0,0

```

```

StartSprite1:
dc.w $A05A,$A480 ;HSTART=$B4, VSTART=$A0, VSTOP=$A4, AT=1
dc.w %0000 0000 0000 0000,%0000 0000 0000 0000
dc.w %0000 1111 0000 1100,%0000 0000 1111 1100
dc.w %0011 1111 1111 1100,%0011 1111 1111 1100
dc.w %0000 0000 0000 0000,%0000 0000 0000 0000
dc.w 0,0

```

Mehrere Sprites über einen DMA-Kanal

Nachdem ein Sprite dargestellt wurde, ist dieser DMA-Kanal wieder frei. In dem obigen Beispiel werden die letzten Sprite-Daten in der Zeile 163 gelesen. Danach wird der Sprite-DMA-Kanal durch die beiden Nullen in der Datenliste abgeschaltet. Wie schon erwähnt, ist es aber auch möglich, den DMA-Kanal weiter zu benutzen. Man schreibt hierzu zwei neue Kontrollworte anstelle der beiden Nullen in die Datenliste. Einzige Bedingung hierbei ist, daß zwischen der ersten Zeile des nächsten Sprites und der letzten des vorangegangenen Sprites mindestens

eine Zeile frei bleibt. Geht der vorherige z.B. bis Zeile 163, kann der nächste nicht vor Zeile 165 beginnen. Der Grund dafür ist, daß in der Zeile dazwischen (164) die beiden Kontrollworte gelesen werden müssen. Der Ablauf des Sprite-DMAs ist dann wie folgt:

Zeile	Daten über den DMA-Kanal
162	vorletzte Zeile des 1. Sprites über diesen Kanal
163	letzte Zeile des 1. Sprites
164	Kontrollworte des 2. Sprites
165	erste Zeile des 2. Sprites
166	zweite Zeile des 2. Sprites

Folgendes Beispiel stellt das dreifarbiges Sprite aus unserer ersten Sprite-Datenliste an zwei verschiedenen Bildschirmpositionen dar:

Start:

;erster Sprite über diesen DMA-Kanal an Zeile 160 (\$A0)

;horizontale Position: 180 (\$B4)

dc.w \$A05A,\$A800 ;HSTART = \$B4, VSTART = \$A0, VSTOP = \$A8

dc.w %0000 0000 0000 0000,%0000 0011 1100 0000

dc.w %0000 0000 0000 0000,%0000 1100 0011 0000

dc.w %0000 0001 1000 0000,%0001 1001 1001 1000

dc.w %0000 0011 1100 0000,%0011 0010 0100 1100

dc.w %0000 0011 1100 0000,%0011 0010 0100 1100

dc.w %0000 0001 1000 0000,%0001 1001 1001 1000

dc.w %0000 0000 0000 0000,%0000 1100 0011 0000

dc.w %0000 0000 0000 0000,%0000 0011 1100 0000

;Jetzt folgt der zweite Sprite über diesen DMA-Kanal

;an Zeile 176 (\$B0), horizontale Position 300 (\$12C)

dc.w \$B096,\$B800 ;HSTART = \$12C, VSTART = \$B0, VSTOP = \$B8

dc.w %0000 0000 0000 0000,%0000 0011 1100 0000

dc.w %0000 0000 0000 0000,%0000 1100 0011 0000

dc.w %0000 0001 1000 0000,%0001 1001 1001 1000

dc.w %0000 0011 1100 0000,%0011 0010 0100 1100

dc.w %0000 0011 1100 0000,%0011 0010 0100 1100

dc.w %0000 0001 1000 0000,%0001 1001 1001 1000

dc.w %0000 0000 0000 0000,%0000 1100 0011 0000

dc.w %0000 0000 0000 0000,%0000 0011 1100 0000

dc.w 0,0 ;Ende der Sprite-Datenliste

Aktivierung der Sprites

Nachdem man eine korrekte Sprite-Datenliste im Chip-RAM aufgebaut und die gewünschten Farben in die Farbtabelle geschrieben hat, muß man dem DMA-Controller noch mitteilen, an welcher Adresse sich diese Datenliste befindet, bevor man den Sprite-DMA einschalten kann. Dafür besitzt jeder Sprite-DMA-Kanal ein Registerpaar, in das man die Anfangsadresse der Datenliste schreiben muß:

SPRxPT-Register (SpritePointer, Zeiger auf Datenliste für Sprite-DMA-Kanal x):

Reg.	Name	Funktion
\$120	SPR0PTH	Zeiger auf die Sprite-Datenliste Bits 16-18
\$122	SPR0PTL	für Sprite-DMA-Kanal 0 Bits 0-15
\$124	SPR1PTH	Zeiger auf die Sprite-Datenliste Bits 16-18
\$126	SPR1PTL	für Sprite-DMA-Kanal 1 Bits 0-15
\$128	SPR2PTH	Zeiger auf die Sprite-Datenliste Bits 16-18
\$12A	SPR2PTL	für Sprite-DMA-Kanal 2 Bits 0-15
\$12C	SPR3PTH	Zeiger auf die Sprite-Datenliste Bits 16-18
\$12E	SPR3PTL	für Sprite-DMA-Kanal 3 Bits 0-15
\$130	SPR4PTH	Zeiger auf die Sprite-Datenliste Bits 16-18
\$132	SPR4PTL	für Sprite-DMA-Kanal 4 Bits 0-15
\$134	SPR5PTH	Zeiger auf die Sprite-Datenliste Bits 16-18
\$136	SPR5PTL	für Sprite-DMA-Kanal 5 Bits 0-15
\$138	SPR6PTH	Zeiger auf die Sprite-Datenliste Bits 16-18
\$13A	SPR6PTL	für Sprite-DMA-Kanal 6 Bits 0-15
\$13C	SPR7PTH	Zeiger auf die Sprite-Datenliste Bits 16-18
\$13D	SPR7PTL	für Sprite-DMA-Kanal 7 Bits 0-15

(alle SPRxPT können nur beschrieben werden)

Der DMA-Controller benutzt diese Register als Zeiger auf die aktuelle Adresse in der Sprite-Datenliste. Zum Beginn jedes Bildes enthalten sie die Adresse des ersten Kontrollworts. Mit jedem gelesenen Datenwort werden sie um ein Wort erhöht, so daß sie am Ende des Bildes auf das erste Wort nach der Datenliste zeigen. Damit in jedem Bild dieselben Sprites dargestellt werden, müssen diese Zeiger vor jedem Bild wieder auf den Anfang der Sprite-Datenliste zurückgesetzt werden. Wie schon bei den

Bitplane-Zeigern BPLxPT erledigt dies am einfachsten der Copper innerhalb der vertikalen Austastlücke. Der entsprechende Abschnitt der Copper-Liste kann etwa so aussehen:

StartSpriteH = Anfangsadresse der Sprite-Datenliste für Sprite x, Bits 16-19

StartSpriteL = Bits 0-15

```
CopperlistStart
MOVE #StartSprite0H,SPR0PTH ;Initialisierung von Sprite-DMA-
MOVE #StartSprite0L,SPR0PTL ;Kanal0
MOVE #StartSprite1H,SPR1PTH ;Initialisierung von Sprite-DMA-
MOVE #StartSprite1L,SPR1PTL ;Kanal1
MOVE #StartSprite2H,SPR2PTH ;Kanal2
MOVE #StartSprite2L,SPR2PTL
... .. ;ebenso für Kanal 3 bis 6
MOVE #StartSprite7H,SPR4PTH ;zuletzt noch Kanal 7
MOVE #StartSprite7L,SPR4PTL
... .. ;Sonstige Aufgaben des Coppers
WAIT $FFFE ;Ende der Copperlist
```

Es gibt keine Möglichkeit, die Sprite-DMA-Kanäle einzeln ein- und auszuschalten. Das SPREN-Bit (Bit Nr. 5) im DMACON-Register schaltet den Sprite-DMA für alle acht Sprite-DMA-Kanäle ein. Will man nicht alle davon verwenden, muß man die unbenutzten Kanäle leere Datenlisten bearbeiten lassen. Dazu setzt man ihre SPRxPT auf zwei Speicherworte mit dem Inhalt Null. Man kann dazu z.B. die beiden Nullen am Ende einer vorhandenen Datenliste benutzen.

Es müssen aber immer alle acht SPRxPT innerhalb der vertikalen Austastlücke initialisiert werden! Auch wenn die Datenliste nichts außer den zwei Nullen enthält, zeigt der SPRxPT des DMA-Kanals am Ende eines Bilds auf das erste Wort nach ihnen.

Selbstverständlich kann die Initialisierung der SPRxPT auch vom Prozessor innerhalb des Vertical-Blanking-Interrupts erledigt werden.

Als letzter Schritt muß noch der Sprite-DMA eingeschaltet werden. Dies geschieht, wie schon erwähnt, mittels des SPREN-Bit des DMACON-Registers für alle acht Sprite-DMA-Kanäle gemeinsam. Folgender Move-Befehl erledigt dies:

```
MOVE.W #$8220,$DFF096 ;SPREN und DMAEN im DMACON-Register setzen
```

Bewegen von Sprites

Die Position eines Sprites bestimmen die Werte der beiden Kontrollworte in der Sprite-Datenliste. Um ein Sprite zu bewegen, muß man diese Werte schrittweise verändern. Dies kann direkt vom Prozessor mittels entsprechender Move-Befehle erledigt werden. Man muß allerdings etwas aufpassen. Ändert man die Kontrollworte zu einem beliebigen Zeitpunkt, kann folgendes Problem auftauchen:

Der Prozessor modifiziert das erste Kontrollwort. Bevor er das zweite ebenfalls verändern kann, liest der DMA-Controller beide Worte. Da diese jetzt nicht mehr zueinander passen, erscheint kurzzeitig irgendwelcher Unsinn auf dem Bildschirm.

Man kann das am einfachsten vermeiden, indem man die Kontrollworte lediglich innerhalb der vertikalen Austastlücke ändert (innerhalb des Vertical-Blanking-Interrupts, nachdem der Copper die SPRxPT initialisiert hat).

Die Sprite/Playfield-Priorität

Die Priorität eines Playfields oder Sprites bestimmt, ob es vor, hinter oder zwischen den anderen Bildelementen erscheint. Das Sprite mit der höchsten Priorität befindet sich vor allen anderen. Es kann nicht von ihnen verdeckt werden. Die Priorität eines Sprites wird durch seine Nummer festgelegt. Je niedriger, desto höher ist die Priorität. Sprite 0 steht damit immer vor allen anderen Sprites.

Bei den Playfields bestimmt ein Kontrollbit, ob Nummer 1 oder 2 vorne liegt.

Wie aber steht es mit der Priorität der Sprites in bezug auf die Playfields?

Beim Amiga ist es möglich, die Playfields fast beliebig zwischen den Sprites zu positionieren. Bei der Festlegung der Playfield-Priorität gegenüber den Sprites werden immer zwei Sprites zusammengefaßt und gemeinsam behandelt. Es sind dieselben Kombinationen wie schon bei den fünfzehnfarbigen Sprites. Immer ein Sprite mit einer geraden Nummer und sein ungerader Nachfolger:

Sprite 0 & 1, Sprite 2 & 3, Sprite 4 & 5, Sprite 6 & 7

Man kann die vier Sprite-Paare als einen Stapel aus vier Elementen betrachten. Blickt man von oben auf einen solchen Stapel, kann man darunterliegende Elemente nur durch Löcher in darüberliegenden Stapelpositionen sehen. Die Löcher entsprechen den transparenten Punkten der Bitplanes oder Sprites und den Teilen des Bildschirms, die ein Sprite auf Grund seiner Größe nicht verdecken kann. Die Reihenfolge der Elemente auf dem Stapel ist nicht veränderbar. Aber man kann zwei andere Elemente, nämlich die Playfields, an beliebigen Stellen zwischen den vier Sprite-Paaren einordnen. Fünf Positionen sind für jedes Playfield möglich:

Position	Reihenfolge von vorne nach hinten				
0	PLF	SPR0&1	SPR2&3	SPR4&5	SPR6&7
1	SPR0&1	PLF	SPR2&3	SPR4&5	SPR6&7
2	SPR0&1	SPR2&3	PLF	SPR4&5	SPR6&7
3	SPR0&1	SPR2&3	SPR4&5	PLF	SPR6&7
4	SPR0&1	SPR2&3	SPR4&5	SPR6&7	PLF

Das BPLCON2-Register enthält die Priorität der Playfields in bezug auf die Sprites:

BLPCON2 \$104 (nur schreiben)

Bit Nr.:	15-7	6	5	4	3	2	1	0
Funktion:	unbel.	PF2PRI	PF2P2	PF2P1	PF2P0	PF1P2	PF1P1	PF1P0

PF2PRI

Ist dieses Bit gesetzt, erscheint Playfield 2 vor Playfield 1.

PF1P0 bis PF1P2

Diese drei Bits bilden eine 3-Bit-Zahl, die die Position des Playfields Nummer 1 (alle ungeraden Bitplanes) zwischen den vier Sprite-Paaren bestimmt. Es sind nur Werte von 0 bis 4 möglich (siehe obige Tabelle).

PF2P0 bis PF2P2

Diese drei Bits entsprechen in ihrer Funktion den Bits PF1P0 bis PF1P2, nur diesmal für Playfield Nummer 1 (alle geraden Bitplanes).

Beispiel:

BPLCON2 = \$0003

Die bedeutet Playfield 1 vor Playfield 2, PF2P0-2 = 0, PF1P0-2 = 3. Dies ergäbe, von vorne nach hinten, folgende Reihenfolge:

PLF2 SPR0&1 SPR2&3 SPR4&5 PLF1 SPR6&7

Genau betrachtet, etwas paradox. Das PLF2PRI-Bit ist 0, also Playfield 1 vor Playfield 2. Trotzdem stimmt die Reihenfolge oben. Wenn sich eines der Sprites 0 bis 5 zwischen Playfield 1 und 2 befindet, steht es, entsprechend seiner Priorität, vor Playfield 1. Da dieses vor Playfield 2 steht, ist an dieser Stelle das Sprite sichtbar, obwohl es sich eigentlich hinter Playfield 2 befinden müßte. Liegen dagegen nur Playfield 2 und das Sprite an einer gemeinsamen Position, verdeckt Playfield 2 aufgrund seiner Priorität das Sprite.

Dies liegt daran, daß die Playfield/Playfield-Priorität Vorrang vor der Sprite/Playfield-Priorität hat.

Verwendet man den Dual-Playfield-Modus nicht, gibt es nur ein einziges Playfield, das dann von den geraden und ungeraden

Bitplanes gemeinsam gebildet wird. Das PLF2PRI- und die PL2P0- bis PL2P2-Bits haben dann keine Funktion mehr.

Kollisionen zwischen Grafikelementen

Es ist häufig sehr nützlich zu wissen, ob zwei Sprites miteinander oder mit dem Hintergrund kollidiert sind. Dies erleichtert z.B. in Spielprogrammen die Erkennung eines Treffers.

Wenn sich an einer bestimmten Bildschirmposition die Punkte zweier Sprites überlappen, d.h. beide einen gesetzten Punkt (nicht transparent) an denselben Koordinaten besitzen, wird dies als Kollision zwischen den zwei Sprites aufgefaßt. Auch ein Zusammenstoß der Playfields miteinander oder mit einem Sprite ist möglich.

Jede erkannte Kollision wird in dem Kollisions-Datenregister, CLXDAT, gespeichert:

CLXDAT \$00E (nur lesen)

Bit Nr. Kollision zwischen

15	unbenutzt
14	Sprite 4 (oder 5) und Sprite 6 (oder 7)
13	Sprite 2 (oder 3) und Sprite 6 (oder 7)
12	Sprite 2 (oder 3) und Sprite 4 (oder 5)
11	Sprite 0 (oder 1) und Sprite 6 (oder 7)
10	Sprite 0 (oder 1) und Sprite 4 (oder 5)
9	Sprite 0 (oder 1) und Sprite 2 (oder 3)
8	Playfield 2 (gerade Bitplanes) und Sprite 6 (oder 7)
7	Playfield 2 (gerade Bitplanes) und Sprite 4 (oder 5)
6	Playfield 2 (gerade Bitplanes) und Sprite 2 (oder 3)
5	Playfield 2 (gerade Bitplanes) und Sprite 0 (oder 1)
4	Playfield 1 (ungerade Bitplanes) und Sprite 6 (oder 7)
3	Playfield 1 (ungerade Bitplanes) und Sprite 4 (oder 5)
2	Playfield 1 (ungerade Bitplanes) und Sprite 2 (oder 3)
1	Playfield 1 (ungerade Bitplanes) und Sprite 0 (oder 1)
0	Playfield 1 und Playfield 2

Während bei einem Sprite jeder nichttransparente Punkt eine Kollision auslösen kann, läßt sich bei den Playfields frei einstellen, welche Farbe zur Kollisionserkennung herangezogen werden soll. Außerdem ist es möglich, jedes Sprite mit einer ungeraden Nummer wahlweise in die Kollisionserkennung einzubeziehen oder von ihr auszuschließen. Mit den Bits im Kollisions-Kontrollregister, CLXCON, läßt sich dies alles einstellen.

CLXCON \$098 (nur schreiben)

Bit Nr.	Name	Funktion
15	ENSP7	Kollisionserkennung für Sprite 7 erlauben
14	ENSP5	Kollisionserkennung für Sprite 5 erlauben
13	ENSP3	Kollisionserkennung für Sprite 3 erlauben
12	ENSP1	Kollisionserkennung für Sprite 1 erlauben
11	ENBP6	Bitplane 6 mit MVBP6 vergleichen
10	ENBP5	Bitplane 5 mit MVBP5 vergleichen
9	ENBP4	Bitplane 4 mit MVBP4 vergleichen
8	ENBP3	Bitplane 3 mit MVBP3 vergleichen
7	ENBP2	Bitplane 2 mit MVBP2 vergleichen
6	ENBP1	Bitplane 1 mit MVBP1 vergleichen
5	MVBP6	Wert für Kollision mit Bitplane 6
4	MVBP5	Wert für Kollision mit Bitplane 5
3	MVBP4	Wert für Kollision mit Bitplane 4
2	MVBP3	Wert für Kollision mit Bitplane 3
1	MVBP2	Wert für Kollision mit Bitplane 2
0	MVBP1	Wert für Kollision mit Bitplane 1

Die ENSPx-Bits (Enable Sprite x) bestimmen, ob das entsprechende Sprite mit ungerader Nummer zur Kollisionserkennung herangezogen wird. Ist z.B. das ENSP1-Bit gesetzt, wird eine Kollision zwischen Sprite 1 und einem anderen Sprite oder einem Playfield registriert. Dabei wird dasselbe Bit im Kollisions-Datenregister gesetzt, wie für Sprite 0. Es ist also nicht möglich, anhand des Registerinhalts zu entscheiden, ob Sprite 0 oder 1 eine Kollision ausgelöst hat. Außerdem werden keine Kollisionen von Sprite 0 und Sprite 1 untereinander erkannt. Bei der Auswahl der Sprites sollte man diese Tatsachen beachten.

Hat man zwei Sprites zu einem fünfzehnfarbigen Sprite kombiniert, muß das entsprechende ENSPx-Bit gesetzt werden, um eine korrekte Kollisionserkennung zu ermöglichen.

Bei den Playfields kann man selber festlegen, welche Bitkombinationen der Bitplanes eine Kollision auslösen sollen und welche nicht. Die ENBPx-Bits (Enable Bitplane x) bestimmen, welche Bitplanes zur Kollisionserkennung herangezogen werden. Sind alle ENBPx-Bits eines Playfields gesetzt, ist eine Kollision an allen Punkten möglich, deren Bitkombination derjenigen der MVBPx-Bits (Match Value Bitplane x) entspricht.

Die ENBPx-Bits bestimmen, ob die Bits aus Plane x mit dem Wert von MVBPx verglichen werden. Stimmen bei einem Punkt die Bits aller Planes, bei denen ENBPx-gesetzt ist, mit dem zugehörigen MVBPx überein, kann dieser Punkt eine Kollision auslösen.

Kompliziert? Ein Beispiel schafft Klarheit:

Die ENBPx-Bits sind gesetzt, ebenso alle MVBPx-Bits. Jetzt können nur noch diejenigen Punkte des Playfields eine Kollision auslösen, deren Bitkombination binär 111111 beträgt. Sind nur die unteren drei MVBPx-Bits gesetzt, ist eine Kollision lediglich möglich, wenn der Punkt des Playfields die Kombination 000111 hat.

Soll eine Kollision an allen Punkten mit der Bitkombination 000111, 000110, 000100 oder 000101 erlaubt sein, müssen die MVBP-Bits auf 000100 stehen. Die unteren beiden Bits sollen immer der Kollisionsbedingung erfüllen, dazu werden die entsprechenden ENBPx-Bits gelöscht, ENBP-Wert: 111100.

Beispiele für mögliche Bitkombinationen:

ENBPx	MVBPx	Kollision möglich bei Bitmuster
111111	111111	111111
111111	111000	111000
111100	1111xx	111100, 111101, 111110, 111111
011111	x00000	000000, 100000
000000	xxxxxx	Kollision bei jedem Bitmuster möglich

(Der Wert eines mit "x" bezeichneten Bits ist irrelevant)

Sind nicht alle sechs Bitplanes aktiv, müssen die ENBPx-Bits der unbenutzten Planes auf 0 gesetzt werden!

Die verschiedenen Kombinationsmöglichkeiten der ENBPx- und MVBPx-Bits erlauben eine Vielfalt unterschiedlicher Kollisionserkennungen. Man kann z.B. das CLXCON-Register so einstellen, daß die Sprites nur mit den roten und grünen Punkten des Playfields zusammenstoßen können, nicht aber mit den anderen Farben. Oder eine Kollision ist nur an den transparenten Punkten von Playfield 1 möglich, wenn die dahinterliegenden Punkte von Playfield 2 gleichzeitig schwarz sind usw.

Sonstige Sprite-Register

Für jedes Sprite existieren neben den SPRxPT-Registern noch vier weitere Register. Sie werden normalerweise vom DMA-Controller automatisch mit Daten versorgt. Es ist aber auch möglich, sie von Hand, d.h. mit dem Prozessor, anzusprechen.

SPRxPOS	erstes Kontrollwort
SPRxCTL	zweites Kontrollwort
SPRxDATA	erstes Datenwort einer Zeile (Low-Wort)
SPRxDATB	zweites Datenwort einer Zeile (High-Wort)

(x steht wieder für eine Sprite-Nummer von 0 bis 7, die Adressen dieser Register finden sich in der Registerübersicht in Kapitel 1.5.1)

Der DMA-Controller schreibt die beiden Kontrollworte eines Sprites direkt in zwei Register, SPRxPOS und SPRxCTL. Wenn ein Wert in das SPRxCTL-Register geschrieben wird, egal ob per DMA oder vom 68000, schaltet Denise die Sprite-Ausgabe ab. Das Sprite wird nicht mehr auf dem Bildschirm ausgegeben.

Der DMA-Controller wartet jetzt auf die in VSTART festgelegte Zeile. Dann schreibt er die zwei ersten Datenworte in das SPRxDATA- und SPRxDATB-Register. Damit wird das Sprite angezeigt, denn beim Schreiben in das SPRxDATA-Register, schaltet Denise die Sprite-Ausgabe wieder ein. Es vergleicht ab jetzt die gewünschte horizontale Position aus den SPRxCTL- und SPRxPOS-Registern mit der tatsächlichen Bildschirmspalte und stellt das Sprite an der richtigen Stelle auf dem Monitor dar.

Der DMA-Controller schreibt in jeder Zeile zwei neue Datenworte in SPRxDATA/B, bis die letzte Zeile des Sprites (VSTOP) vorbei ist. Danach holt er die nächsten Kontrollworte und platziert sie in SPRxPOS und SPRxCTL. Damit wird das Sprite wieder abgeschaltet, bis die nächste VSTART-Position erreicht ist. Waren beide Kontrollworte Null, beendet der DMA-Controller den Sprite-DMA für den entsprechenden Kanal bis zum Beginn des nächsten Bildes. Am Ende der vertikalen Austastlücke beginnt er wieder an der aktuellen Adresse in SPRxPT.

Darstellung der Sprites ohne DMA

Man kann ein Sprite auch problemlos ohne den DMA-Kanal darstellen. Man schreibt dazu die gewünschten Kontrollworte einfach direkt in die SPRxPOS- und SPRxCTL-Register. Dabei müssen nur die HSTART-Position und das AT-Bit gültige Werte enthalten. VSTART und VSTOP werden ausschließlich vom DMA-Kanal benutzt.

Man kann die Sprite-Ausgabe in jeder beliebigen Zeile beginnen, indem man die beiden Datenworte in das SPRxDATA- und SPRxDATB-Register schreibt. Da SPRxDATA die Sprite-Ausgabe einschaltet, ist es besser, wenn man erst SPRxDATB beschreibt. Ändert man die Daten in beiden Registern nicht, wer-

den sie in jeder Zeile wieder angezeigt. Es entsteht ein vertikaler Balken.

Will man das Sprite wieder abschalten, schreibt man einfach einen beliebigen Wert in SPRxPOS.

1.5.7 Der Blitter

Was ist ein Blitter? Der Name Blitter ist eine Abkürzung der englischen Bezeichnung: "Block Image Transferrer", was etwa soviel heißt wie Bild-Datenblock-Kopierer. Genau dies ist nämlich die Hauptaufgabe des Blitters: das Verschieben und Kopieren von Datenblöcken im Speicher, wobei es sich meistens um Grafikdaten handelt. Der Blitter kann auch mehrere Speicherbereiche miteinander logisch verknüpfen und das Ergebnis wieder zurück in den Speicher schreiben. Er erledigt diese Aufgaben sehr schnell. Einfaches Datenverschieben geht mit einer Geschwindigkeit von bis zu 16 Millionen Punkten in der Sekunde vonstatten!

Zusätzlich kann der Blitter noch Flächen füllen und Linien ziehen. Die Kombination dieser beiden Fähigkeiten ermöglicht das Zeichnen beliebiger, ausgefüllter Vielecke, und das vielfach schneller, als es mit dem 68000 möglich wäre.

Das Betriebssystem verwendet den Blitter für beinahe alle Grafikoperationen. Er erledigt die Textausgabe, zeichnet Gadgets, verschiebt Fenster usw. Außerdem erledigt er auch noch die Decodierung der Daten von der Diskette. Ein Beispiel dafür, daß sich die vielfältigen Fähigkeiten des Blitters nicht nur auf die Grafik beschränken.

Die Verwendung des Blitters zum Kopieren von Daten

Der Blitter arbeitet beim Kopieren von Daten immer nach dem gleichen Muster: Ein bis drei verschiedene Speicherbereiche, die Datenquellen, werden miteinander durch die gewählte logische Bedingung verknüpft, und das Ergebnis wird wieder zurück in den Speicher geschrieben. Das Spektrum reicht vom einfachen

Kopieren bis zum komplexen Verknüpfen mehrerer Datenbereiche. Die Adressen der Quelldatenbereiche, genannt A, B und C, und des Zieldatenbereichs D lassen sich innerhalb des Chip-RAMs frei wählen (Adressen von 0 bis \$7FFFF).

Die Anzahl der Worte, die in einer Blitter-Operation verarbeitet werden, darf bis zu 65536 betragen. Es können also bis zu 128 KByte Daten in einem Zug durch den Speicher geschleudert werden.

Der Blitter unterstützt sogenannte rechteckige Speicherbereiche. D.h. der Speicher ist, wie bei einer Bitmap, in Spalten und Zeilen aufgeteilt. Es ist auch möglich, einen kleinen Bereich innerhalb einer großen Bitmap zu bearbeiten, indem sogenannte Modulo-Werte verwendet werden. Vielleicht erinnern Sie sich daran: Solche Modulo-Werte wurden auch schon bei den Playfields verwendet, um Bitplanes zu definieren, die breiter als das Bildschirmfenster waren.

Um eine Blitter-Operation zu starten, sind folgende Schritte notwendig:

- Wählen des Blitter-Modus: Daten kopieren
- Auswahl der Quelldatenbereiche (Es müssen nicht immer alle drei Quellen benutzt werden.) und des Zielbereichs
- Auswahl der logischen Verknüpfung
- Definition sonstiger Betriebsparameter (Scrolling, Maskierung, Adressierungsrichtung)
- Festlegung des Fensters, in dem die Blitter-Operation durchgeführt werden soll, und Starten des Blitters

Die Festlegung des Blitter-Fensters

Sie werden sich fragen, warum wir mit der Beschreibung des letzten Punkts beginnen. Nun, eigentlich ist die Definition des gewünschten Fensters die Grundlage aller anderen Einstellungen. Aber wenn man den Blitter programmiert, wird dieser Wert erst am Ende in das entsprechende Register geschrieben, da der

Blitter damit gleichzeitig gestartet wird. Aus diesem Grund steht dieser Punkt auch zuletzt in der obigen Liste. Für das Verständnis der anderen Werte ist es aber notwendig, den Begriff "Blitter-Fenster" zu kennen.

Das Blitter-Fenster ist der Speicherbereich, den der Blitter in der Blitter-Operation bearbeiten soll. Er ist wie eine Bitplane aufgebaut, d.h. in Zeilen und Spalten aufgeteilt, wobei eine Spalte einem Wort (2 Bytes) entspricht. Die Anzahl der Worte im Fenster ist also gleich dem Produkt aus Zeilen und Spalten: $Z * S$.

Durch die Aufteilung des gewünschten Speicherbereichs in Zeilen und Spalten ist der Blitter sehr gut für die Bearbeitung von Bitplanes geeignet.

Genausogut kann man aber auch lineare Speicherbereiche ansprechen. Die Aufteilung in Zeilen und Spalten ist ja nur eine Hilfe zur einfacheren Programmierung. In Wirklichkeit liegen die einzelnen Zeilen weiterhin an fortlaufenden Adressen im Speicher. Bei kleinen Datenfeldern, die nicht in Zeilen und Spalten eingeteilt sind, ist es auch möglich, die Fensterbreite oder Höhe auf 1 zu setzen.

Der Blitter arbeitet das Blitter-Fenster zeilenweise ab. Die Blitter-Operation beginnt mit dem ersten Wort in der ersten Zeile und endet beim letzten Wort der letzten Zeile.

Das BLTSIZE-Register enthält die Fenstergröße:

BLTSIZE \$058 (nur schreiben)

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Funktion:	H9	H8	H7	H6	H5	H4	H3	H2	H1	H0	W5	W4	W3	W2	W1	W0

H0-H9 Diese zehn Bits stellen die Höhe (Height) des Blitter-Fensters in Zeilen dar. Das Fenster kann eine Höhe zwischen 1 und 1024 Zeilen haben ($2^{10} = 1024$). Dabei wird eine Höhe von 1024 Zeilen dadurch gewählt, daß man Height auf 0 setzt. Für alle übrigen

Werte entspricht Height direkt der Zeilenzahl. Eine Höhe von 0 Zeilen ist nicht möglich.

W0-W5 Sechs Bits repräsentieren die Breite (Width) des Fensters. Sie kann von 1 bis 64 Worten ($2^6 = 64$) variieren. Rechnet man dies in Grafikpunkte um (1 Wort = 16 Pixels), ergibt das bis zu 1024 Punkte. Wie bei der Höhe erreicht man die maximale Breite von 64 Worten, indem man Width = 0 setzt.

Um aus der Höhe und Breite den notwendigen BLTSIZE-Wert zu ermitteln, dient folgende Formel: $\text{BLTSIZE} = \text{Höhe} * 64 + \text{Breite}$.

Sollen auch die beiden Extremfälle Height = 1024 und Width = 64 erfaßt werden, muß sie etwas modifiziert werden:

$$\text{BLTSIZE} = (\text{Höhe AND } \$3\text{FF}) * 64 + (\text{Breite AND } \$3\text{F})$$

Das BLTSIZE-Register sollte immer als letztes Register initialisiert werden. Der Blitter wird beim Schreiben in BLTSIZE automatisch gestartet.

Quell- und Zieldatenbereiche

Bei einer Blitter-Operation werden Daten aus völlig verschiedenen Speicherbereichen miteinander verknüpft. Auch wenn das Blitter-Fenster die Anzahl und Anordnung der zu bearbeitenden Daten festlegt, muß die Positionierung dieses Fensters innerhalb der drei Quell- und des Zieldatenbereichs noch bestimmt werden.

Nehmen wir z.B. an, der Blitter soll eine kleine, rechteckige Grafik, die irgendwo im Chip-RAM gespeichert ist, in den Bildschirmspeicher kopieren. Bei dieser einfachen Aufgabe gibt es nur einen Quellbereich. Die Wahl des Blitter-Fensters ist einfach. Die gesamte Grafik soll kopiert werden, also entspricht die Breite und Höhe des Blitter-Fensters der der Grafik im Speicher.

Damit der Blitter auch weiß, wo diese Grafik zu finden ist, schreibt man die Adresse des ersten Worts der obersten Zeile in das entsprechende Register.

Wie aber muß man den Zielbereich definieren? Die Grafik soll in den Bildschirmspeicher, sie muß also in die aktuelle Bitplane übertragen werden. (Der Einfachheit halber sollen sowohl Grafik als auch Bildschirmspeicher nur aus einer Bitplane bestehen.) Nun ist die Bitplane aber um ein Vielfaches breiter als die kleine Grafik. Würde der Blitter sie direkt in die Bitplane kopieren, sähe das etwas seltsam aus. Es muß dem Blitter also neben der Adresse des Zielbereichs noch dessen Breite mitgeteilt werden. Dies geschieht mittels sogenannter Modulo-Werte. Der Modulo-Wert wird nach jeder abgearbeiteten Zeile des Blitter-Fensters zu dem Adreßzeiger addiert. Dadurch werden die Worte, die nicht beeinflußt werden sollen, übersprungen, und der Zeiger steht wieder auf dem Anfang der nächsten Zeile. Damit Quell- und Zieldatenbereiche unterschiedlich breit sein können, haben sie voneinander unabhängige Modulo-Register.

Kopieren einer Grafik in eine Bitplane

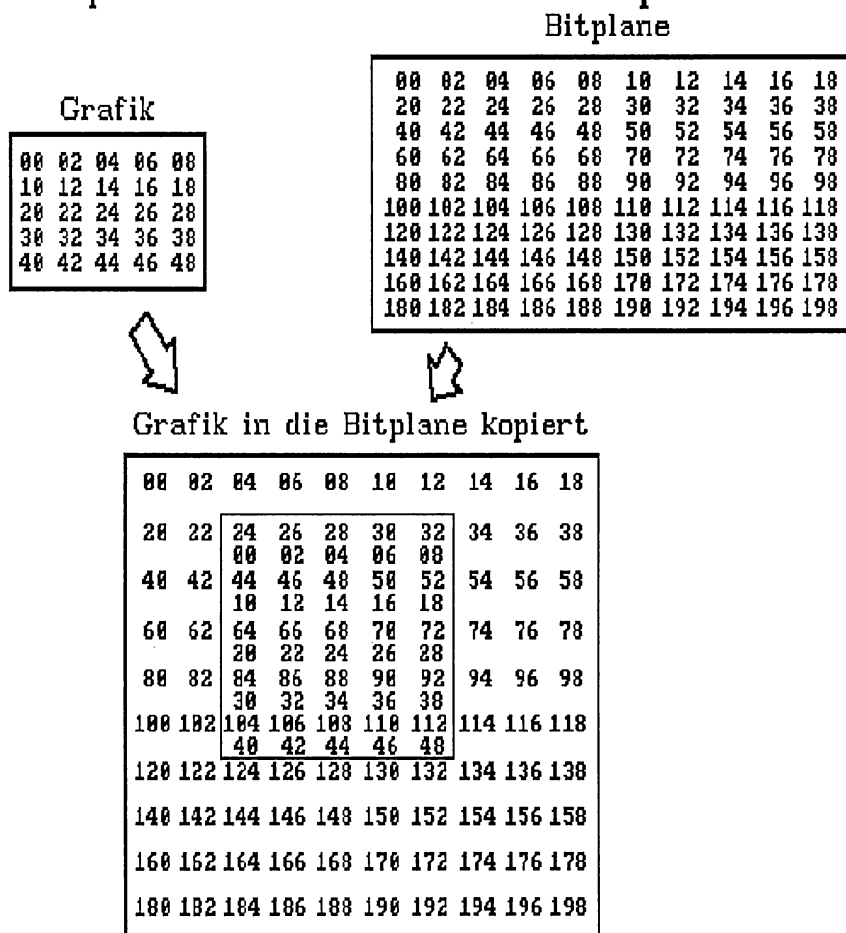


Abb. 1.5.7.1

Die Abbildung 1.5.7.1 illustriert unser Beispiel. Die Grafik besteht aus 5 Zeilen, jede 10 Worte breit. Die Zahlen stehen für die Adresse des entsprechenden Words in bezug auf die Anfangsadresse der Grafik. Die Bitplane hat eine Höhe von 10 Zeilen zu 20 Worten. Wie müssen jetzt Blitter-Fenster, Anfangsadressen und Modulo-Werte gewählt werden?

Das Blitter-Fenster muß der Grafik entsprechen, da diese ja komplett kopiert werden soll, d.h. Höhe des Fensters 5 Zeilen und Breite 10 Worte. Der Wert, der nachher in das BLTSIZE-Register geschrieben werden muß, beträgt also 330 oder hexadezimal \$014A.

Die Anfangsadresse der Quelldaten ist gleich der Adresse des ersten Datenworts der Grafik. Da die Breite einer Zeile der Grafik gleich der Zeilenbreite des Blitter-Fensters ist, bleibt der Modulo-Wert der Quelle auf 0.

Für den Zieldatenbereich muß jetzt der Modulo-Wert bestimmt werden. Dazu bildet man einfach die Differenz aus der tatsächlichen Zeilenbreite und der des Blitter-Fensters. In unserem Beispiel 20 Worte minus 10 Worte: Der Modulo-Wert für den Zieldatenbereich beträgt 10 Worte. In den Modulo-Registern des Blitters muß der Modulo-Wert in Bytes angegeben werden. $\text{Modulo-Wert} = \text{Modulo in Worten} * 2$.

Als letztes benötigt der Blitter noch die Anfangsadresse der Zieldaten. Diese bestimmt die Position, an der die Grafik in die Bitplane kopiert wird, und ist gleich der Anfangsadresse der Bitplane plus der Adresse der Worte, an der die linke, obere Ecke der Grafik plaziert werden soll. In unserer Abbildung ist dies die Adresse der Bitplane plus 24.

Wie läuft die Blitter-Operation ab?

Nachdem die Adressen und die Modulo-Werte unseres Beispiels feststehen, beginnt der Blitter nach der Initialisierung von BLTSIZE mit dem Kopieren der Daten. Er holt das Wort an der Anfangsadresse der Quelldaten und speichert es an der Zieladresse ab. Danach addiert er ein Wort zu beiden Adressen dazu und kopiert das nächste Wort. Dies wird sooft wiederholt, bis die in BLTSIZE eingestellte Anzahl von Worten pro Zeile bearbeitet wurde. Bevor der Blitter jetzt mit der nächsten Zeile fortfährt, addiert er den Modulo-Wert zu den Adreßzeigern, damit die nächste Zeile an der richtigen Adresse beginnt.

Nachdem alle Zeilen kopiert worden sind, schaltet sich der Blitter ab und wartet auf den nächsten Auftrag.

Die Adreßregister enthalten nach einer Blitter-Operation die Adresse des letzten Worts plus 2 und plus dem Modulo-Wert.

Die Adreßregister heißen BLTxPT, wobei x für eine der drei Quellen A, B, C oder den Zieldatenbereich D steht. Die Adreßregister bestehen wie üblich aus einem Register für die Bits 0-15 und einem für die Bits 16-18:

Reg.	Name	Funktion
048	BLTCPTH	Anfangsadresse des Bits 16-18
04A	BLTCPTL	Quelldatenbereich C Bits 0-15
04C	BLTBPTH	Anfangsadresse des Bits 16-18
04E	BLTBPTL	Quelldatenbereich B Bits 0-15
050	BLTAPTH	Anfangsadresse des Bits 16-18
052	BLTAPTL	Quelldatenbereich A Bits 0-15
054	BLTDPH	Anfangsadresse des Bits 16-18
056	BLTDPTL	Zieldatenbereich D Bits 0-15

Jeder der vier Bereiche besitzt ein eigenes Modulo-Register:

060	BLTCMOD	Modulo-Wert für Quelle C
062	BLTBMOD	Modulo-Wert für Quelle B
064	BLTAMOD	Modulo-Wert für Quelle A
066	BLTDMOD	Modulo-Wert für Zieldaten D

Kopieren mit aufsteigenden oder absteigenden Adressen

In unserem Beispiel arbeitete der Blitter mit aufsteigenden Adressen, d.h. er beginnt bei der Anfangsadresse und erhöht diese schrittweise bis zur Endadresse. Die Endadresse ist dabei logischerweise höher als die Anfangsadresse.

Es gibt aber einen Fall, in dem eine solche Adressierung zu Fehlern führt: Das Kopieren eines Speicherbereichs an eine höhere Adresse, wobei sich Quell- und Zielbereich teilweise überlappen. Dazu ein Beispiel:

Adresse	Quelldaten	Zielfdaten	Ergebnis	
			gewünscht	tatsächlich
0	Quell1			
2	Quell2			
4	Quell3			
6	Quell4	Ziel1	Quell1	Quell1
8	Quell5	Ziel2	Quell2	Quell2
10		Ziel3	Quell3	Quell3
12		Ziel4	Quell4	!Quell1!
14		Ziel5	Quell5	!Quell2!

Die fünf Quelldatenworte sollen an die Adresse der Zielfdaten geschrieben werden. Wenn der Blitter bei Quell1 beginnt, überschreibt er Quell4, wenn er Quell1 an der gewünschten Zielfadresse Ziel1 abspeichert, da Quell4 und Ziel1 dieselbe Adresse haben (beide Bereiche überlappen sich). Das gleiche passiert bei Quell2 und Ziel2.

Wenn der Blitter dann bei der Adresse von Quell4 ankommt, befindet sich dort statt dessen Quell1. So landet Quell1 anstelle von Quell4 in Ziel4 und Quell2 in Ziel5, während Quell4 und 5 verlorengehen.

Als Lösung dieses Problems besitzt der Blitter neben dem Ascending Mode (aufsteigende Adressierung) noch den Descending Mode (absteigende Adressierung). In dieser Betriebsart beginnt er bei den Adressen in BLTxPT und erniedrigt sie nach jedem kopierten Wort um 2 Bytes. Auch der Modulo-Wert wird nicht addiert, sondern subtrahiert. Die Endadresse liegt also vor der Anfangsadresse.

Dies muß man natürlich bei der Initialisierung der BLTxPT berücksichtigen. Normalerweise setzt man sie auf die linke, obere Ecke des Blitter-Fensters im jeweiligen Datenbereich (A, B, C oder D). Im Descending Mode verläuft die Adressierung rückwärts. Entsprechend muß BPLxPT auf die rechte, untere Ecke zeigen.

Die Modulo- und die BLTSIZE-Werte sind mit denen des Ascending Mode identisch.

Allgemein kann man über die Wahl des richtigen Modi folgende Aussagen machen:

1. Keine Überlappung zwischen Quell- und Zielbereich:
Entweder Ascending oder Descending Mode, beide arbeiten in diesem Fall korrekt.
2. Quell- und Zielbereich überlappen sich teilweise, wobei der Zielbereich vor dem Quellbereich liegt:
Nur der Ascending Mode arbeitet korrekt.
3. Quell- und Zielbereich überlappen sich teilweise, wobei der Zielbereich hinter dem Quellbereich liegt (s. Beispiel):
Nur der Descending Mode arbeitet korrekt.

Die Auswahl der logischen Verknüpfungen

Wie erwähnt, gibt es drei Quelldaten, die zu den Zieldaten verknüpft werden. Diese logischen Verknüpfungen laufen immer nur bitweise ab, d.h. aus den drei Datenbits A, B und C muß das Zielbit D gewonnen werden.

Der Blitter kennt 256 verschiedene Verknüpfungen. Diese entstehen in zwei Stufen:

1. Es werden acht verschiedene boolsche Gleichungen auf die drei Datenbits angewandt. Jede davon liefert bei einer anderen Kombination aus A, B und C eine 1 als Ergebnis.
2. Die acht Ergebnisse obiger Gleichungen werden wahlweise miteinander durch ein logisches ODER verknüpft. Das Ergebnis ist das Zielbit D.

Der Begriff "Boolesche Gleichung" steht für einen mathematischen Ausdruck, der eine Kombination logischer Verknüpfungen darstellt. Diese Rechenweise nennt man Boolesche Algebra, nach dem englischen Mathematiker George Boole (1815 bis 1864). Die weiteren Erläuterungen der logischen Funktionen des Blitters sind auch ohne Kenntnisse der Booleschen Algebra zu verstehen. Die Booleschen Gleichungen werden aber mit aufgeführt.

Bei drei Bit gibt es 8 mögliche Kombinationen. Jede der acht Gleichungen ist bei einer davon wahr (ihr Ergebnis ist 1). Mittels acht Kontrollbits LF0 bis LF7 kann man wählen, ob das Ergebnis der Gleichung zur Bildung von D herangezogen werden soll. Alle Ergebnisbits, deren zugehöriges LFX-Bit auf 1 ist, werden miteinander logisch ODER-verknüpft. Eine ODER-Verknüpfung bedeutet, daß das Ergebnis 1 ist, wenn mindestens ein Eingangsbit ebenfalls auf 1 steht. Anders ausgedrückt, ein logisches ODER liefert nur eine 0, wenn alle Eingänge 0 sind.

Mittels der acht LFX-Bits kann man also wählen, bei welchen Kombinationen der drei Eingangsbits A, B, C das Ausgangsbit D gleich 1 ist.

Die englische Bezeichnung für die acht Booleschen Eingangsgleichungen lautet "Minterms".

Folgende Tabelle gibt einen Überblick über die zu jedem LFX-Bit gehörige Eingangskombination.

(In der Spalte Miniterm steht ein kleiner Buchstabe für eine NOT-Verknüpfung des entsprechenden Eingangsbits. Üblicherweise wird dies durch einen Querstrich über dem Buchstaben dargestellt.)

Die Spalte "Eingangsbits" enthält die Bitkombination, für die die entsprechende Gleichung erfüllt ist. Die Reihenfolge der Bits ist gleich A B C.

	LF7	LF6	LF5	LF4	LF3	LF2	LF1	LF0
Miniterm:	ABC	ABc	AbC	Abc	aBC	aBc	abC	abc
Eingangsbits:	111	110	101	100	011	010	001	000

Die Auswahl der einzelnen Miniterme ist einfach. Man setzt alle LFx-Bits, bei deren zugehöriger Eingangskombination das Ausgangsbit D gleich 1 sein soll.

In unserem ersten Beispiel sollten die Quelldaten von A direkt nach D kopiert werden. Die Quellen B und C werden nicht verwendet. Welche Miniterme müssen dafür ausgewählt werden?

D darf nur 1 sein, wenn $A = 1$ ist. Es kommen also nur die oberen vier Terme LF4 bis LF7 in Frage, da nur bei ihnen $A = 1$ ist. Damit B keine Rolle spielt, wählt man zwei Terme, in denen B einmal 1 und einmal 0 ist, die sonst aber identisch sind. Jetzt hat B keine Auswirkung mehr auf D, denn der Rest der Gleichung bleibt für beide möglichen Fälle von B unverändert, und ihr Ergebnis ist nur noch von diesem Rest abhängig. Das gleiche gilt für C. Betrachtet man die Tabelle der Eingangskombinationen, sieht man, daß LF4 bis LF7 aktiviert sein müssen. Dann hängt das Ergebnis nur noch von A ab, denn je nach Kombination von B und C ist immer eine dieser vier Gleichungen für $A = 1$ wahr und D damit gleich 1. Ist $A = 0$, sind alle vier unwahr und $D = 0$.

Wer sich mit der Booleschen Algebra auskennt, kann auch ganz formal zu den notwendigen Minitermen kommen. Der benötigte Ausdruck ist $A = D$. Da B und C beim Blitter immer mit dabei sind, müssen sie entsprechend in die Gleichung integriert werden:

$$A * (b+B) * (c+C) = D$$

Der Term $x+X$ ist immer wahr (gleich 1) und wird verwendet, wenn der Wert von x für das Ergebnis D gleichgültig sein soll. Um jetzt zu den notwendigen Minitermen zu kommen, muß man nur noch ausmultiplizieren:

1. $A*(b+B)*(c+C)=D$
2. $(A*b+A*B)*(c+C)=D$
3. $A*b*c+A*B*c+A*b*C+A*B*C=D$

Wie üblich kann man die Mal-Zeichen weglassen:

$$Abc+ABc+AbC+ABC=D$$

Jetzt braucht man nur noch die LFX-Bits der entsprechenden Miniterme zu setzen. Wie man sieht, kommt man mit der Booleschen Algebra ebenfalls zum Ziel.

Einige Beispiele häufig benötigter Blitter-Operationen und der zugehörigen LFX-Bits:

- Invertieren eines Datenbereichs: $a = D$

Notwendige LFX-Kombination: 00001111

Boolesche Algebra: $a = D$

$$a*(b+B)*(c+C) = D$$

$$(ab+aB)*(c+C) = D$$

$$abc+aBc+abC+aBC = D$$

- Kopieren einer Grafik in eine Bitplane, ohne deren Inhalt zu verändern; entspricht einem logischen ODER der Grafik A mit der Bitplane B: $A + B = D$

Notwendige LFX-Kombination: 11111100

Boolesche Algebra: $A + B = D$

$$A(b+B)(c+C)+B(a+A)(c+C) = D$$

$$(Ab+AB)(c+C)+(Ba+BA)(c+C) = D$$

$$Abc+ABc+AbC+ABC+Bac+BAc+BaC+BAC = D$$

$$Abc+ABc+AbC+ABC+aBc+aBC = D$$

Zuletzt noch einmal die Regel zum Heraussuchen der notwendigen LFX-Bits:

1. Bei welchen der acht Kombinationen von ABC soll D gleich 1 werden?
2. Die LFX-Bits der entsprechenden Kombinationen setzen.
3. Werden nicht alle drei Quellen benötigt, müssen sämtliche Kombinationen gewählt werden, in denen die unbenutzten Bits vorkommen und die gewünschten Bits den richtigen Wert besitzen.

Verschieben der Eingangswerte

Bei manchen Aufgaben stört es, daß der Blitter an Wortgrenzen gebunden ist. So kann es z.B. vorkommen, daß ein bestimmter Bereich innerhalb einer Bitmap um einige Punkte verschoben werden soll. In diesem Fall müssen die Datenbits nur um den Teil eines Wortes bewegt werden. Oder der Blitter soll eine Grafik an einer bestimmten Koordinate in den Bildschirmspeicher schreiben, die nicht mit einer Wortgrenze übereinstimmt.

Um diese Probleme bewältigen zu können, hat der Blitter die Fähigkeit, die Datenworte aus den Quellen A und B um bis zu 15 Bits nach rechts zu verschieben. Damit ist er in der Lage, die Daten an jede gewünschte Punktposition zu bringen. Alle Bits, die bei dem Verschiebevorgang nach rechts hinausgeschoben werden, gelangen beim nächsten Wort in die dort freiwerdenden Bits. Es wird quasi die gesamte Zeile bitweise verschoben. Das Verschieben übernimmt im Blitter ein sogenannter Barrel-Shifter. Er benötigt für einen Verschiebevorgang keine zusätzliche Zeit, egal, um wie viele Bits verschoben werden soll. Eine zusätzliche Verschiebung der Daten schränkt die Geschwindigkeit des Blitters also in keiner Weise ein!

Beispiel für eine Verschiebung der Daten um 3 Bit:

Vorher:

Datenwort 1	Datenwort 2	Datenwort 3
00011111 10011100	00010101 01111111	11100001 11100101

Nachher:

Datenwort 1	Datenwort 2	Datenwort 3
xxx00011 11110011	10000010 10101111	11111100 00111100

Die drei xxx-Bits hängen von dem vorangegangenen Datenwort ab, aus dem sie hinausgeschoben wurden.

Maskierung

Es ist möglich, daß der Blitter eine Grafik aus dem Bildschirmspeicher herauskopieren soll, deren Ränder nicht auf Wortgrenzen liegen. Die Daten, die sich links vom dem Rand der Grafik, aber noch innerhalb des ersten Datenworts befinden, sollen nicht mitkopiert werden. Um dies zu ermöglichen, hat der Blitter die Fähigkeit, das erste und letzte Datenwort einer Zeile mit einer Maske zu behandeln. Dies bedeutet, daß man wählen kann, welche Bits dieser Wörter übernommen werden. Dadurch kann man unerwünschte Daten an den Rändern der Zeile löschen.

Die Maskierungsmöglichkeit existiert nur für Quelle A. Zwei Register enthalten die Masken für die beiden Ränder. Nur wenn ein Bit im Maskenregister gesetzt ist, wird es bei der Blitter-Operation übernommen. Alle übrigen werden gelöscht.

\$044 BLTAFWM Blitter Source A First Word Mask

Maske für das erste Datenwort in der Zeile.

\$046 BLTALWM Blitter Source A Last Word Mask

Maske für das letzte Datenwort in der Zeile.

Die Bits 0 - 15 enthalten die entsprechenden Maskenbits.

Beispiel:

("1" steht für ein gesetztes Bit, "." für ein gelöschtches)

Grafikdaten in der Bitplane:

Spalte 1	Spalte 2	Spalte 3
.....11111111	1111111111111111	1.....11
111111.....1111	11.....1111	1111.....1111
....11.....11	1111.....111	11111.....11111111
....11.....1	11111.....11	1111111111111111
....11.....1	11111.....11	1111111111111111
....11.....11	1111.....111	11111....11111111
111111.....1111	11.....1111	1111.....1111
.....11111111	1111111111111111	1.....11

FirstWordMask:
0000000011111111

LastWordMask:
1111110000000000

Ergebnis:

Spalte 1	Spalte 2	Spalte 3
.....11111111	1111111111111111	1.....
.....1111	11.....1111	1111.....
.....11	1111.....111	11111.....
.....1	11111.....11	111111.....
.....1	11111.....11	111111.....
.....11	1111.....111	11111.....
.....1111	11.....1111	1111.....
.....11111111	1111111111111111	1.....

Durch Ausmaskieren der unerwünschten Bildelemente an den Rändern erhält man die gewünschte Grafik.

Achtung!

Wenn die Breite des Blitter-Fensters lediglich ein Wort beträgt (BLTSIZE.Width = 1), fallen beide Masken zusammen. Sie wirken gemeinsam auf das Eingangswort. Nur die Eingangsbits, deren Maskenbits in beiden Masken gesetzt sind, werden durchgelassen.

Die Blitter-Kontrollregister

Der Blitter besitzt zwei Kontrollregister, BLTCON0 und BLITCON1. In diesen Registern befinden sich verschiedene Kontrollbits zur Blitter-Steuerung:

BLTCON0 \$040

Bit Nr.	Name	Funktion
---------	------	----------

15	ASH3	ASH0-3 enthalten den Wert für die Verschiebung der Eingangsdaten von Quelle A
14	ASH2	
13	ASH1	ASH0-3 = 0 bewirkt keine Verschiebung
12	ASH0	
11	USEA	Schaltet den DMA-Kanal für Quelle A ein
10	USEB	Schaltet den DMA-Kanal für Quelle B ein
9	USEC	Schaltet den DMA-Kanal für Quelle C ein
8	USED	Schaltet den DMA-Kanal für Ziel D ein
7	LF7	Wählt Miniterm ABC (Bitkomb. von ABC: 111)
6	LF6	Wählt Miniterm ABc (Bitkomb. von ABC: 110)
5	LF5	Wählt Miniterm AbC (Bitkomb. von ABC: 101)
4	LF4	Wählt Miniterm Abc (Bitkomb. von ABC: 100)
3	LF3	Wählt Miniterm aBC (Bitkomb. von ABC: 011)
2	LF2	Wählt Miniterm aBc (Bitkomb. von ABC: 010)
1	LF1	Wählt Miniterm abC (Bitkomb. von ABC: 001)
0	LF0	Wählt Miniterm abc (Bitkomb. von ABC: 000)

BLTCON1 \$042

Bit Nr.	Name	Funktion
---------	------	----------

15	BSH3	BSH0-3 enthalten den Wert für die Verschie-
14	BSH2	bung der Eingangsdaten von Quelle B
13	BSH1	BSH0-3 = 0 bewirkt keine Verschiebung
12	BSH0	
1 - 5		unbelegt
4	EFE	Exclusive Fill Enable
3	IFE	Inclusive Fill Enable
2	FCI	Fill Carry In
1	DESC	DESC = 1 schaltet auf Descending Mode
0	LINE	LINE = 1 aktiviert den Linienmodus

Das LINE-Bit schaltet den Blitter auf das Zeichnen von Linien um. Will man den Blitter zum Kopieren von Daten benutzen, muß LINE = 0 sein.

Mit dem DESC-Bit kann zwischen aufsteigenden und absteigenden Adressen umgeschaltet werden. Ist DESC = 0, arbeitet der Blitter im Ascending-, bei DESC = 1 im Descending Mode.

Die EFE- und IFE-Bits aktivieren die Betriebsart "Flächen füllen" des Blitters. Sie müssen beide auf 0 sein, wenn der Blitter normal betrieben werden soll.

Das FCI-Bit hat nur im Zusammenhang mit dem Füllmodus eine Funktion.

Der Blitter-DMA

Die Daten der Quellbereiche A, B und C und die Ausgangsdaten D werden mittels vier DMA-Kanälen aus dem Speicher gelesen bzw. in diesen hineingeschrieben. Diesen Blitter-DMA kann man mit dem BLTEN (Bit 6) des DMACON-Registers für alle Kanäle gemeinsam einschalten. Der Blitter besitzt für seine DMA-Transfers vier Datenregister:

Adr.	Name	Funktion
000	BLTDDAT	Ausgangsdaten D
070	BLTCDAT	Datenregister von Quelle C
072	BLTBDAT	Datenregister von Quelle B
074	BLTADAT	Datenregister von Quelle A

Der DMA-Controller liest die benötigten Eingangswerte aus dem Speicher und schreibt sie in die Datenregister. Hat der Blitter die Eingangsdaten verarbeitet, enthält BLTDDAT das Ergebnis. Der DMA-Controller überträgt den Inhalt von BLTDDAT in das Chip-RAM.

Mittels der drei USEx-Bits läßt sich der DMA-Transfer über diese vier Register ein- und ausschalten. USEA = 0 schaltet z.B. den DMA-Kanal für das Datenregister A ab. Der Blitter greift aber weiterhin auf den Wert in BLTADAT zu, d.h. mit jedem neuen Wort der aktiven Quellen wird immer dasselbe Wort aus Quelle A geholt. Aus diesem Grund muß man bei nicht benutzten Quellen USEx = 0 setzen und zusätzlich noch durch eine entsprechende Auswahl der Miniterme jeden Einfluß dieser Quelle auf das Ergebnis ausschließen (s. oben). Eine andere Möglichkeit ist die bewußte Ausnutzung der Tatsache, daß nach Abschaltung des DMA-Kanals immer dasselbe Datenwort verwendet wird. Man kann damit z.B. den Speicher mit einem Muster füllen, das man direkt mit dem Prozessor in BLTxDAT hineingeschrieben hat.

Außer BLTEN gehören noch drei weitere Bits im DMACON-Register zum Blitter:

Bit 10 BLTPRI

Dieses Bit wurde schon im Kapitel Grundlagen erklärt. Ist es auf 1, hat der Blitter absolute Priorität über den Prozessor.

Bit 14 BBUSY (Dieses Bit kann man nur lesen.)

BBUSY signalisiert den Zustand des Blitters. Ist es auf 1, führt er gerade eine Operation aus.

Nach dem Setzen des Blitter-Fensters in BLTSIZE beginnt der Blitter mit seinem DMA und setzt BBUSY, bis das letzte Wort des eingestellten Blitter-Fensters bearbeitet und wieder zurück in den Speicher geschrieben wurde. Er beendet dann seinen DMA und löscht BBUSY.

Gleichzeitig mit dem Löschen von BBUSY wird auch noch das Blitter-Finished-Bit im Interrupt-Request Register gesetzt.

Bit 13 BZERO

Das BZERO-Bit zeigt an, ob bei einer Blitter-Operation sämtliche Ergebnisbits null waren. Mit anderen Worten, BZERO ist gesetzt, wenn bei allen Datenworten keine der gewählten Verknüpfungen eine 1 als Ergebnis lieferte. Mit Hilfe des BZERO-Bits kann man z.B. eine Kollisionsüberprüfung durchführen. Man setzt die Miniterme so, daß D nur 1 wird, wenn beide Quellen ebenfalls 1 sind. Überschneiden sich die Grafiken in beiden Quellen auch nur an einem Punkt, ist das Ergebnis 1, und BZERO wird gelöscht. Am Ende der Blitter-Operation kann man so feststellen, ob eine Kollision vorlag oder nicht. USED setzt man dabei auf 0, damit die Ausgangsdaten nicht in den Speicher geschrieben werden.

Die Verwendung des Blitters zum Füllen von Flächen

Was hat man sich unter dem Begriff "Füllen von Flächen" vorzustellen?

Unter einer Fläche versteht der Blitter einen zweidimensionalen Speicherbereich, den er mit Punkten füllen soll. Normalerweise gehört diese Fläche zu einer Grafik oder einer Bitplane.

Um eine Fläche auffüllen zu können, muß der Blitter ihre Begrenzungen kennen. Eine für den Blitter verständliche Definition

der Grenzlinien ist notwendig. Vielseitige Füllfunktionen existieren in den meisten Malprogrammen und auch im Amiga-BASIC beim PAINT-Befehl. Bei ihnen wird ein Bereich des Bildschirms, ausgehend von einem Startpunkt, ausgefüllt, bis das Programm auf eine Begrenzungslinie stößt. Damit lassen sich völlig frei gewählte Flächen ausmalen, vorausgesetzt, daß sie von einer durchgehenden Linie eingeschlossen sind. Der Blitter ist nicht in der Lage, eine solch komplexe Fülloperation auszuführen. Er arbeitet immer nur zeilenweise und füllt dabei die freien Plätze zwischen zwei gesetzten Bits aus, die die Grenzen der gewünschten Fläche festlegen. Zwei Beispiele zeigen den Ablauf eines Füllvorgangs des Blitters:

Fehlerfreie Fülloperation:

Vorher:	Nachher:
.....1.1.....	111.....
.....1.....1.....	11111111.....
.....1.....1.....	111111111111.....
....1...1...1...1.....	111111...111111.....
....1...1...1...1.....	111111...111111.....
.....1.....1.....	111111111111.....
.....1.....1.....	11111111.....
.....1.....1.....	111111111111.....

Fehlerhafte Fülloperation auf Grund falsch gewählter Begrenzungsbits:

Vorher:	Nachher:
.....111.....	111111111111.....
.....111...111.....	111111111.....
.....11...111...11.....	11111111111111...11.....
....1...1...1...1.....	111111...111111.....
....1...1...1...1.....	111111...111111.....
.....11...111...11.....	11111111111111...11.....
.....1.....1.....	111111111.....
.....111111111111.....	11111111111111111111.....

Beim ersten Beispiel wurde eine Fläche Blitter-gerecht abgegrenzt und auch korrekt gefüllt. Anders dagegen Nummer 2. Hier hat man eine geschlossene Umgrenzungslinie um die Figur

gezeichnet. Versucht man eine solche Grafik mit dem Blitter zu füllen, entsteht ein Chaos.

Grund dafür ist der Algorithmus des Blitters. Er ist äußerst einfach. Der Blitter beginnt an der rechten Seite der Zeile. Um zu bestimmen, ob ein Bit gesetzt werden muß, benutzt er das sogenannte FillCarry-Bit (FC). Normalerweise ist es am Anfang gleich 0. Der Blitter prüft jetzt den Wert des Bits rechts außen. Ist es auf Null, bleibt der Wert des FC-Bits erhalten. Es bildet das entsprechende Bit der Ausgangsdaten. Der Blitter fährt in dieser Weise mit den Nachbarbits fort, bis er auf ein gesetztes Eingangsbit stößt. Dadurch wird das FC-Bit auf 1 gesetzt. Da die Ausgangsbits dem aktuellen Wert des FC-Bit entsprechen, sind sie ebenfalls 1. Erst wenn der Blitter erneut ein gesetztes Eingangsbit findet, löscht er das FC-Bit wieder. Auf diese Weise wird immer der Bereich zwischen zwei gesetzten Bits aufgefüllt. Wie man aus dem zweiten Beispiel sehen kann, kommt die Fülllogik bei einer ungeraden Anzahl von Bytes etwas durcheinander.

Den Startwert des FC-Bits bestimmt das FCI-Bit (FillCarryIn) im BLTCON1. Ist FCI gelöscht, läuft alles wie oben beschrieben ab. Ist FCI = 1, beginnt der Blitter vom Rand her zu füllen, bis er auf das erste gesetzte Eingangsbit stößt. Der Füllvorgang läuft umgekehrt ab.

Beispiel der Auswirkung des FCI-Bits:

Ausgangsgrafik	FCI=0	FCI=1
.....1.....1.....	1111111.....	1111111.....111111
....1.....1.....1...	111111111111...	11111.....1111
...1.....1.1.....1..	...111111.111111..	1111....111....111
...1.....1.1.....1..	...111111.111111..	1111....111....111
....1.....1.....1...	111111111111...	11111.....1111
.....1.....1.....	1111111.....	1111111.....111111

Bei den bisherigen Beispielen blieben die Eingangsbits, also die Randbegrenzungen, im aufgefüllten Bild erhalten. Dies ist immer der Fall, wenn man den Füllmodus durch Setzen des ICE-Bits (Inclusive Fill Enable) im BLTCON1-Register aktiviert.

Im Gegensatz dazu steht der ECE-Modus (Exclusive Fill Enable), der durch das Setzen des gleichnamigen Bits in BLTCON1 eingeschaltet wird. Bei ihm werden die Begrenzungsbits am linken Rand einer gefüllten Fläche (also immer, wenn das Fill-Carry-Bit von 1 auf 0 wechselt) nicht in das Ausgangsbild übernommen. Damit werden sämtliche Flächen um einen Punkt schmaler. Nur im ECE-Modus ist es möglich, Flächen mit der Breite von nur einem Bit zu erhalten. Dies ist im ICE-Modus nicht möglich, denn zur Markierung einer Fläche, sei sie auch noch so schmal, sind mindestens zwei Begrenzungsbits notwendig, die dann auch beide im Ausgangsbild erscheinen.

Unterschied zwischen dem ICE- und ECE-Modus:

Ausgangsbild	ICE	ECE
.....11.....11	11.....11	1.....1
.....1...1....1.1	11111...111	1111.....11
...1...11..1..1..1	...111111111..1111	...1111..111...111
1.....11...11...1	111111111111111111	.1111111..1111..1111
..1.....1	..1111111111111111	...1111111111111111
....1.....1	11111111111111	11111111111111
.....1...1...11..	11111...11..	1111....1..

Bitweiser Ablauf der verschiedenen Fülloperationen:

Eingangsmuster: 11010010

Bit Nr.	Eing. Bit	FCI = 0			FCI = 1		
		FC	ICE	ECE	FC	ICE	ECE
-	11010010	FC=FCI			FC=FCI		
0	0	0	0	0	1	1	1
1	1	1	10	10	0	11	01
2	0	1	110	110	0	011	001
3	0	1	1110	1110	0	0011	0001
4	1	0	11110	01110	1	10011	10001
5	0	0	011110	001110	1	110011	110001
6	1	1	1011110	1001110	0	1110011	0110001
7	1	0	11011110	01001110	1	11110011	10110001

(FC=FCI bedeutet, daß das FC-Bit vor Beginn des Füllvorgangs den Wert des FCI-Bit aus BLTCON1 annimmt.)

Wie wird eine Fülloperation des Blitters gestartet?

Der Blitter kann diesen Füllvorgang zusätzlich zu einem normalen Kopiervorgang ausführen. Eingeschaltet wird er, indem man je nach gewünschtem Modus entweder das ICE- oder ECE-Bit in BLTCON1 setzt. Der Blitter bildet wie immer aus den drei Quellen A, B, C über die gewählten Miniterme die Ausgangsdaten D. Ist keiner der beiden Füllmodi aktiv, übernimmt der Blitter diese Daten direkt in sein Ausgangsdatenregister (BLTDDAT, \$000), von wo sie per DMA in den Speicher gelangen, wenn USED = 1 ist.

Im Füllmodus dagegen werden die Ausgangsdaten D als Eingangsdaten der Füllschaltung verwendet. Das Ergebnis der Fülloperation wird dann ins Ausgangsdatenregister BLTDDAT geschrieben.

Um eine Fülloperation auszuführen, sind folgende Schritte notwendig:

- Die BLTxPT, BLTxMOD und die Miniterme so auswählen, daß die Ausgangsdaten D die korrekten Begrenzungsbits der zu füllenden Fläche enthalten.
- Descending Mode wählen (Der Blitter füllt von rechts nach links, dies funktioniert nur, wenn auch die Wörter mit absteigenden Adressen angesprochen werden.).
- Den gewünschten Füllmodus wählen: ICE oder ECE setzen; FCI nach Wunsch setzen oder löschen.
- LINE = 0 (Linienmodus aus)
- BLTSIZE auf die Größe der zu füllenden Grafik setzen.

Der Blitter beginnt jetzt mit dem Füllvorgang. Wenn er damit fertig ist, setzt er wie üblich BLTBUSY auf 0.

Die Geschwindigkeit des Blitters wird durch Aktivieren des Füllmodus nicht eingeschränkt. Im Maximalfall ist der Blitter in der Lage, Flächen mit einer Geschwindigkeit von 16 Millionen Punkten pro Sekunde zu füllen. Hauptanwendungsgebiet des

Füllmodus ist das Zeichnen ausgefüllter Vielecke. Dazu wird in einem leeren Speicherbereich mittels des Linienmodus das gewünschte Vieleck eingezeichnet. Der Blitter füllt dies dann mit hoher Geschwindigkeit auf.

Die Verwendung des Blitters zum Zeichnen von Linien

Der Blitter ist ein extrem vielseitiges Hilfsmittel. Neben den hervorragenden Fähigkeiten beim Kopieren von Daten und Füllen von Flächen besitzt er einen leistungsfähigen Modus zum Zeichnen von Linien. Wie auch die anderen Blitter-Modi ist er außerordentlich schnell: bis zu einer Million Punkte pro Sekunde. Selbst mit einem 68020 Prozessor kann man diese Geschwindigkeit softwaremäßig nur mit Mühe erreichen. Für den eingebauten 68000 ist es völlig unmöglich.

Was bedeutet eigentlich "Zeichnen von Linien"?

Beim Zeichnen von Linien sollen zwei Punkte durch eine Gerade miteinander verbunden werden. Da die Auflösung einer Computergrafik begrenzt ist, kann nicht immer der optimale Punkt gewählt werden. Die tatsächlichen Punkte liegen immer etwas über oder unter der gedachten Idealgeraden. Eine solche Linie ähnelt meist mehr oder weniger einer Treppe. Je höher die Auflösung, desto kleiner sind die Stufen, aber man kann sie niemals vollkommen beseitigen.

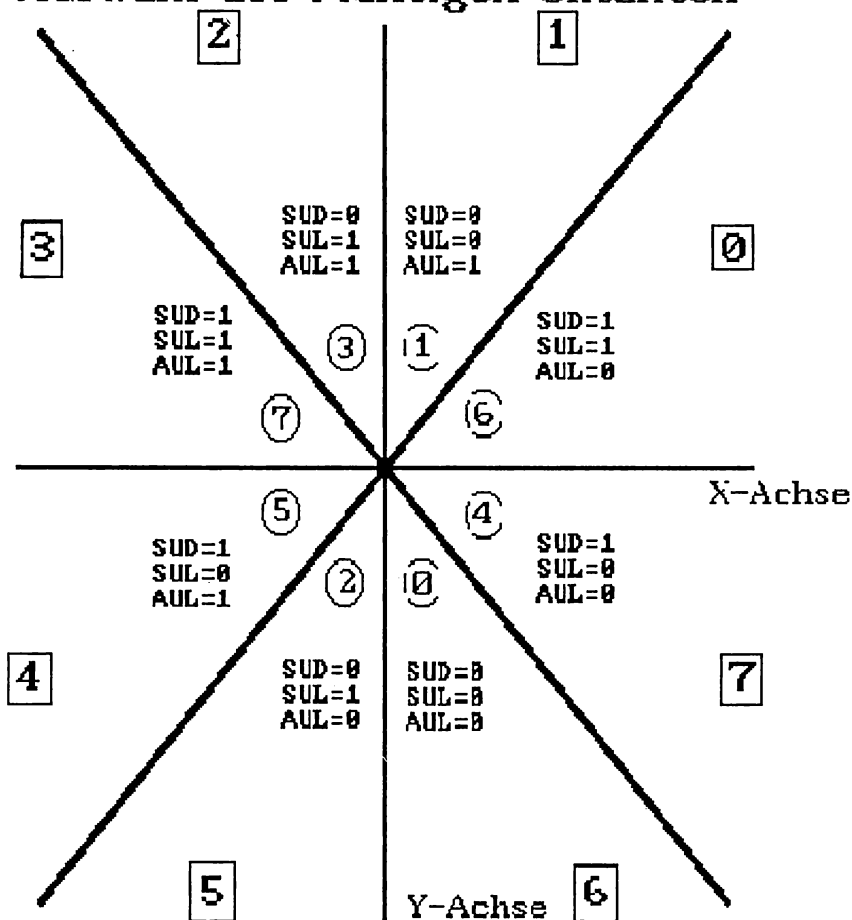
Beispiel für eine Linie in einer Computergrafik:

Die beiden Punkte	werden durch eine Gerade verbunden
.....	
.....1...	111...
.....	111.....
.....	1111.....
.....	111.....
.....1.....	111.....
.....	

Der Blitter ist in der Lage, solche Linien bis zu einer Länge von 1024 Punkten zu zeichnen. Leider kann man nicht einfach die Koordinaten der beiden Endpunkte angeben. Man muß die Linie wieder blittergerecht definieren.

Als erstes benötigt der Blitter dazu den sogenannten Oktanten, in dem die Linie liegt. Die Aufteilung des Koordinatenkreuzes in acht Teile, die sogenannten Oktanten, findet man bei vielen Grafikprozessoren wieder.

Auswahl des richtigen Oktanten



Nummer des Oktanten



Wert der SUD/SUL/AUL-Bits

Abb. 1.5.7.2

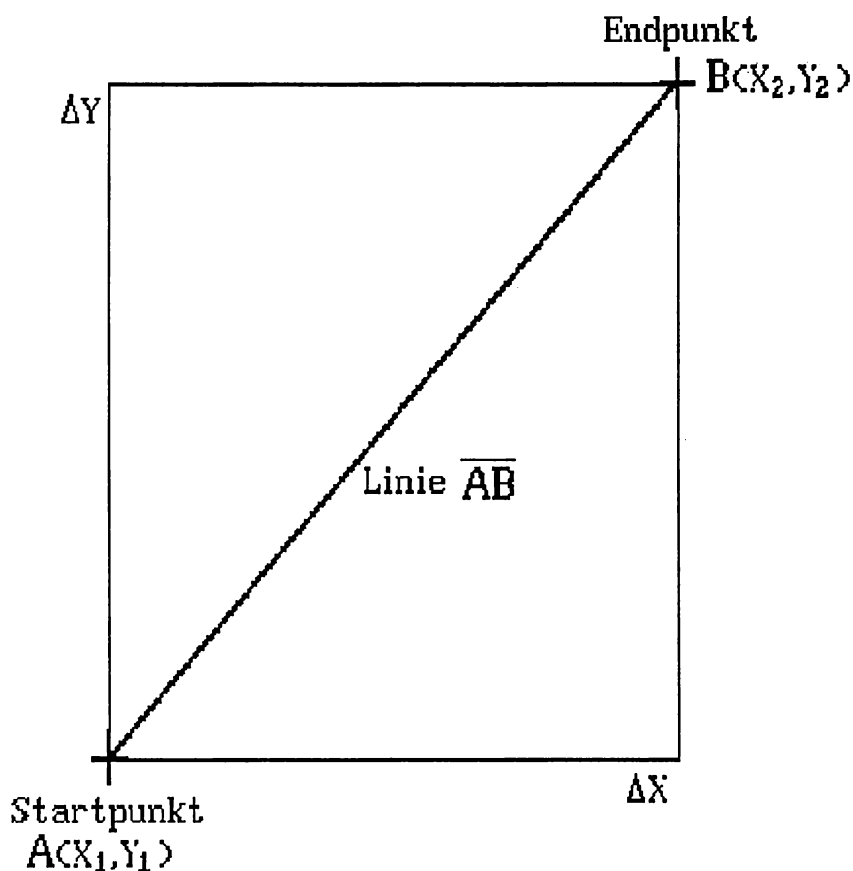


Abb. 1.5.7.3

Die Abbildung 1.5.7.2 zeigt diese Einteilung. Der Startpunkt der Linie liegt im Ursprung des Koordinatenkreuzes (im Schnittpunkt der X- und Y-Achse). Der Endpunkt liegt je nach seinen Koordinaten in einem der acht Oktanten. Mittels drei logischer Vergleiche lässt sich die Nummer dieses Oktanten ermitteln, dabei sind X_1 und Y_1 die Koordinaten des Startpunktes, X_2 und Y_2 die des Endpunktes:

Wenn X_1 kleiner X_2 ist, liegt der Endpunkt in einem der Oktanten 0, 1, 6 oder 7, ist X_1 größer X_2 , sind es Nummer 2, 3, 4 und 5. Sind X_1 und X_2 gleich, liegt er auf der Y-Achse. Dann sind alle acht Oktanten möglich.

Ebenso gilt: Y_1 kleiner Y_2 , möglicher Oktant des Endpunkts: 0, 1, 2 oder 3, wenn Y_1 größer Y_2 ist, sind es die Oktanten 4, 5, 6 oder 7. $Y_1 = Y_2$: alle Oktanten.

Beim letzten Vergleich bildet man die Beträge der X- und Y-Differenz: $\Delta X = |X_2 - X_1|$, $\Delta Y = |Y_2 - Y_1|$. Wenn ΔX größer ΔY ist, kann er in einem der Oktanten 0, 3, 4 oder 7 liegen. Bei ΔX kleiner ΔY befindet er sich in einem der Oktanten 1, 2, 5 oder 6. $\Delta X = \Delta Y$: alle Oktanten.

Der Endpunkt liegt in dem Oktanten, der in allen drei Vergleichen vorkam. Befindet sich der Punkt auf einer der Grenzlinien zwischen zwei Oktanten, ist es egal, welchen von beiden man auswählt.

Auswahl des richtigen Oktanten:

Punktkoordinaten	Oktant	Code	Punktkoordinaten	Oktant	Code
$Y_1 \leq Y_2$ $X_1 \leq X_2$ $\Delta X \geq \Delta Y$	0	6	$Y_1 \geq Y_2$ $X_1 \geq X_2$ $\Delta X \geq \Delta Y$	4	5
$Y_1 \leq Y_2$ $X_1 \leq X_2$ $\Delta X \leq \Delta Y$	1	1	$Y_1 \geq Y_2$ $X_1 \geq X_2$ $\Delta X \leq \Delta Y$	5	2
$Y_1 \leq Y_2$ $X_1 \geq X_2$ $\Delta X \leq \Delta Y$	2	3	$Y_1 \geq Y_2$ $X_1 \leq X_2$ $\Delta X \leq \Delta Y$	6	0
$Y_1 \leq Y_2$ $X_1 \geq X_2$ $\Delta X \geq \Delta Y$	3	7	$Y_1 \geq Y_2$ $X_1 \leq X_2$ $\Delta X \geq \Delta Y$	7	4

Die Ziffern in der Spalte "Code" entsprechen den eingekreisten Ziffern in Abbildung 1.5.7.2. Der Blitter benötigt je nach dem

Oktanten, in dem sich der Endpunkt der Linie befindet, eine spezielle Kombination dreier Bits. Sie heißen SUD (Sometimes Up or Down), SUL (Sometimes Up or Left) und AUL (Always Up or Left). "Code" ist die von diesen Bits gebildete 3-Bit-Zahl (SUD = MSB und AUL = LSB).

Beim Programmieren der Linie muß man also erst den Oktanten des Endpunktes ermitteln und danach den zugehörigen Code-Wert in den Blitter schreiben.

Linien mit Mustern

Der Blitter verwendet beim Zeichnen einer Linie eine Maske, um zu bestimmen, ob die Punkte der Linie gesetzt, gelöscht oder mit einem Muster versehen werden sollen. Die Maske hat eine Breite von 16 Bit, daher wiederholt sich das Muster alle 16 Punkte. Den Zusammenhang zwischen Muster und Aussehen der Linie kann man am besten durch ein paar Beispiele verdeutlichen:

("." = 0, "1" = 1, A = Startpunkt und B = Endpunkt)

Ausgangsbild: Maske = "1111111111111111":

.....1111111.....B.	1111111.....11B.
.....111.....111.....	111.....11111.....
.....11.....11.....	11.....11.11.....
....11.....11....	11.....111....11....
....11.....11....	11.....111.....11....
....11.....11....	11111.....11....
.....111.....111.....	11111.....111.....
..A.....1111111.....	..A11.....1111111.....

Normale Linie:	Linie mit einem Punkt/Zeile:
.....1111	1...
.....1111....	1.....
.....1111.....	1.....
....1111.....	1.....
1111.....	1.....

Die Definition der Steigung

Damit der Blitter weiß, wohin er seine Linie zeichnen soll, benötigt er eine blittergerechte Definition der Liniensteigung. Im einzelnen sind es die Ergebnisse dreier Terme, die alle auf den DeltaX und DeltaY-Werten basieren, wie sie im Abschnitt über die Oktanten erklärt wurden (DeltaY und DeltaX stellen die Breite und Höhe des Rechtecks dar, dessen Diagonale die Linie bildet (siehe Abbildung 1.5.7.3)).

Zuerst muß man beide Werte miteinander vergleichen, um den größeren bzw. kleineren von beiden zu kennen. Nennen wir das kleinere Delta "Kdelta" und das größere "Gdelta". Dann lauten die drei vom Blitter benötigten Ausdrücke folgendermaßen:

1. $2 * Kdelta$
2. $2 * Kdelta - Gdelta$
3. $2 * Kdelta - 2 * Gdelta$

Außerdem besitzt der Blitter noch ein sogenanntes SIGN-Flag, welches auf 1 gesetzt werden muß, wenn $2 * Kdelta < Gdelta$ ist.

Registerfunktionen im Linienmodus

Der Blitter verwendet beim Zeichnen von Linien dieselben Register wie beim Kopieren von Daten (er hat ja nicht mehr), ihre Funktion ändert sich allerdings:

BLTAPTL In BLTAPTL muß der Wert des Ausdrucks " $2 * Kdelta - Gdelta$ " geschrieben werden.

BLTCPT & BLTDPT	Diese beiden Registerpaare (BLTCPTH und BLTCPTL, BLTDPTH und BLTDPTL) müssen mit der Startadresse der Linie initialisiert werden. Das ist die Adresse des Wortes, in dem der Startpunkt der Linie liegt.
BLTAMOD	In BLTAMOD muß der Ausdruck " $2 \cdot K_{\text{delta}} - 2 \cdot G_{\text{delta}}$ " stehen.
BLTBMOD	" $2 \cdot K_{\text{delta}}$ "
BLTCMOD & BLTDMOD	In diese beiden Modulo-Register muß die Breite des gesamten Bildes, in das die Linie soll, geschrieben werden. Dies geschieht wie üblich in Form einer geraden Anzahl Bytes. Bei einer normalen Bitplane mit 320 Punkten (40 Bytes) in X-Richtung wäre der Wert für BLTCMOD bzw. BLTDMOD = 40.
BLTSIZE	Die Breite (Width, Bits 0 bis 5) muß fest auf 2 gesetzt werden. Die Höhe (Height, Bits 6 bis 15) enthält die Länge der Linie in Punkten. Eine Höhe von 0 entspricht einer Linie mit einer Länge von 1024 Punkten. Die korrekte Linienlänge ist immer genau gleich dem Wert von Gdelta. Das Zeichnen der Linie wird durch Schreiben in das BLTSIZE-Register gestartet. Es sollte daher immer als letztes Register gesetzt werden.
BLTADAT	Dieses Register muß mit \$8000 initialisiert werden.
BLTBDAT	BLTBDAT enthält die Maske, mit der die Linie gezeichnet wird.
BLTAFWM	In dieses Maskenregister kommt \$FFFF.

BLTCON0

Bit Nr.	Name	Funktion
15	START3	Der 4-Bit-Wert START0-3 enthält die Position des Startpunkts
14	START2	der Linie innerhalb des Worts
13	START1	an der Startadresse der Linie (BLTCPT/BLTDPT)
12	START0	Normalerweise sind dies die vier unteren Bits der X-Koordinate des Startpunkts.
11	USEA = 1	Diese Kombination der USEx-Bits ist im
10	USEB = 0	Linienmodus notwendig.
9	USEC = 1	
8	USED = 1	
7	LF7	Die LFX-Bits müssen mit \$CA initialisiert
bis 0	LF0	werden ($D = aC + AB$)

BLTCON1

Bit Nr.	Name	Funktion
15	Texture3	Dies ist der Wert für die Verschiebung der
14	Texture2	Maske. Normalerweise setzt man Texture0-3
13	Texture1	= Start0-3. Das Muster im Maskenregister
12	Texture0	BLTBDAT
		beginnt dann mit dem ersten Punkt der
		Linie.
11 bis 7 = 0		unbenutzt, immer auf 0 setzen
6	SIGN	Wenn $2 * K_{\Delta} < G_{\Delta}$ ist, dann muß SIGN
		auf 1
5	---	unbenutzt, immer auf 0 setzen
4	SUL	Diese drei Bits müssen mit dem
		SUL/SUD/AUL-Code des
3	SUD	entsprechenden Oktanten initialisiert
2	AUL	werden (Abbildung 1.5.7.2).
1	SING = 1	Zeichnet Linien mit nur einem Punkt pro
		Zeile.
0	LINE = 1	Schaltet den Blitter auf Linienmodus um.

Zum Abschluß noch ein Zahlenbeispiel:

Es soll eine Linie in eine Bitplane gezeichnet werden. Die Bitplane ist 320 auf 200 Punkte groß und liegt an der Adresse \$40000. Der Startpunkt der Linie hat die Koordinaten $X=20$ und $Y=185$. Der Endpunkt liegt bei $X=210$ und $Y=35$ (Die Koordinaten beziehen sich auf die linke, untere Ecke der Bitplane). $\Delta X = 190$, $\Delta Y = 150$

1. Schritt: Oktant des Endpunktes heraussuchen

Dafür werden die drei entsprechenden Vergleiche durchgeführt, Ergebnis: $X_1 < X_2$, $Y_1 > Y_2$ und $\Delta X > \Delta Y$. Dies ergibt den Oktanten Nr. 7 und einen Wert für den SUD/SUL/AUL-Code von 4.

2. Schritt: Adresse des Startpunkts

Diese errechnet sich wie folgt:

$$\text{Anfangsadresse der Bitplane} + (\text{Anzahl der Zeilen} - Y_1 - 1) * \text{Bytes pro Zeile} + 2 * (X_1 / 16)$$

(Die Nachkommastellen der Division werden abgeschnitten.)

Nach dem Einsetzen der Zahlen:

$$\$40000 + (200 - 185 - 1) * 40 + 2 = \$40232$$

Dieser Wert kommt in BLTCPT und BLTDPT. Die Anzahl der Bytes pro Zeile wird außerdem noch in die BLTCMOD- und BLTDMOD-Register geschrieben.

3. Schritt: Anfangspunkt der Linie in START0-3

Nötige Rechnung: $X_1 \text{ AND } \$F$

In Zahlen: $\text{START0-3} = 20 \text{ AND } \$F = 4$

4. Schritt: Werte für BLTAPTL, BLTAMOD und BLTBMOD

$\Delta Y < \Delta X$, d.h. $K\Delta = \Delta Y$ und $G\Delta = \Delta X$

$$\text{BLTAPTL} = 2 * K\Delta - G\Delta = 2 * 150 - 190 = 110$$

$BLTAMOD = 2 * Kdelta - 2 * Gdelta = 2 * 150 - 2 * 190 = -80$

$BLTBMOD = 2 * Kdelta = 300$

$2 * Kdelta > Gdelta$, daher ist $SIGN = 0$

5. Schritt: Länge der Linie für *BLTSIZE*

Länge = $Gdelta = DeltaX = 190$.

Der Wert des *BLTSIZE*-Registers ergibt sich aus der üblichen Formel: Länge*64 + Breite. "Breite" muß beim Ziehen von Linien immer auf 2 gesetzt werden. $BLTSIZE = DeltaX * 64 + 2 = 12162$ oder $\$2F82$.

6. Schritt: Zusammenfassung der Werte für die beiden *BLTCONx*-Register

In *BLTCON0* muß der *START*-Wert an die richtige Position gebracht werden, dazu kommt *\$CA* für die *LFx*-Bits und 1011 für *USEx*. In unserem Beispiel ergibt dies zusammen *\$ABCA*.

BLTCON1 enthält den Code für den Oktanten und die Kontrollbits. Unsere Linie soll ganz normal gezeichnet werden, also ist *SING* = 0. *LINE* muß natürlich auf 1. *SIGN* wurde schon ausgerechnet und ist in unserem Beispiel ebenfalls 0. Zusammen macht das *\$0011*.

In der Assembler-Sprache könnte die Initialisierung der Register wie folgt aussehen:

```
LEA $DFF00,A5 ;Basisadresse der Customchips nach A5
MOVE.L #$40232, BLTCPTH(A5) ;Startadresse nach BLTCPT
MOVE.L #$40232, BLTDPTH(A5) ;und nach BLTDPT
MOVE.W #40, BLTCMOD(A5) ;Breite der Bitplane nach BLTCMOD
MOVE.W #40, BLTDMOD(A5) ;und nach BLTDMOD
MOVE.W #110, BLTAPTL(A5)
MOVE.W #-80, BLTAMOD(A5)
MOVE.W #300, BLTBMOD(A5)
MOVE.W #$ABCA, BLTCON0(A5)
MOVE.W #$11, BLTCON1(A5)
MOVE.W #12162, BLTSIZE(A5) ;Jetzt beginnt der Blitter mit
                           ;dem Zeichen der Linie
```

Andere Zeichenmodi

Bis jetzt wurde als Wert für die LFX-Bits immer \$CA angegeben. Dies hat zur Folge, daß die Punkte der Linie in Abhängigkeit von der Maske gesetzt oder gelöscht werden, während die übrigen Punkte erhalten bleiben.

Aber es sind auch andere sinnvolle LFX-Kombinationen denkbar. Um dies zu verstehen, muß man wissen, wie die LFX-Bits im Linienmodus interpretiert werden:

Der Blitter kann den Speicher immer nur wortweise adressieren. Im Linienmodus gelangen die Eingangsdaten über den Quellkanal C in den Blitter. Dazu kommt die Maske im B-Register. Welcher Punkt innerhalb des gelesenen Worts der Linienpunkt ist, bestimmt das A-Register. Es enthält ja immer genau ein gesetztes Bit, das vom Blitter an die richtige Position geschoben wird. Der normale LFX-Wert von \$CA bewirkt, daß alle Bits, bei denen das A-Bit gleich 0 ist, direkt von Quelle C übernommen werden. Ist A dagegen gleich 1, wird als Zielbit das entsprechende Maskenbit verwendet.

Weiß man, wie die LFX-Bits verwendet werden, kann man auch andere Zeichenmodi anwählen. \$4A bewirkt beispielsweise, daß alle Linienpunkte invertiert werden.

Die Blitter-DMA-Zyklen

Wie schon im Kapitel Grundlagen besprochen, belegt der Blitter nur die geraden Buszyklen. Da er dabei Priorität über den 68000 hat, ist es interessant zu wissen, wie viele Zyklen dem Prozessor noch bleiben. Dies hängt von der Anzahl der aktiven Blitter-DMA-Kanäle (A, B, C und D) ab. Folgende Tabelle zeigt den Ablauf einer Blitter-Operation für alle fünfzehn möglichen Kombination aktiver und inaktiver Blitter-DMA-Kanäle. Die Buchstaben A, B, C und D stehen dabei für den entsprechenden DMA-Kanal. Die Ziffer "1" dahinter steht für das erste Wort der Blitter-Operation, "2" für alle mittleren und "3" für das letzte Datenwort. Ein Strich "--" bedeutet, daß dieser Buszyklus nicht vom Blitter belegt ist.

Belegung der geraden Buszyklen durch den Blitter:

Aktive DMA-Kanäle	Belegung der geraden Buszyklen
keine	-- -- -- -- -- -- -- -- -- -- -- -- -- --
D	D0 -- D1 -- D2 -- -- -- -- -- -- -- -- -- --
C	C0 -- C1 -- C2 -- -- -- -- -- -- -- -- -- --
C D	C0 -- -- C1 D0 -- C2 D1 -- D2 -- -- -- --
B	B0 -- -- B1 -- -- B2 -- -- -- -- -- -- -- --
B D	B0 -- -- B1 D0 -- B2 D1 -- D2 -- -- -- --
B C	B0 C0 -- B1 C1 -- B2 C2 -- -- -- -- -- --
B C D	B0 C0 -- -- B1 C1 D0 -- B2 C2 D1 -- D2
A	A0 -- A1 -- A2 -- -- -- -- -- -- -- -- -- --
A D	A0 -- A1 D0 A2 D1 -- D2 -- -- -- -- -- --
A C	A0 C0 A1 C1 A2 C2 -- -- -- -- -- -- -- --
A C D	A0 C0 -- A1 C1 D0 A2 C2 D1 -- D2 -- -- --
A B	A0 B0 -- A1 B1 -- A2 B2 -- -- -- -- -- --
A B D	A0 B0 -- A1 B1 D0 A2 B2 D1 -- D2 -- -- --
A B C	A0 B0 C0 A1 B1 C1 A2 B2 C3 -- -- -- -- --
A B C D	A0 B0 C0 -- A1 B1 C1 D0 A2 B2 B3 D1 D2

Anmerkungen:

Obige Tabelle ist nur gültig, wenn folgende Bedingungen erfüllt sind:

1. Der Blitter wird nicht durch Copper- oder Bitplane-DMA-Zugriffe gestört.
2. Der Blitter läuft im Normalmodus (Weder werden Linien gezeichnet, noch werden Flächen gefüllt).
3. Das BLITPRI-Bit im DMACON-Register ist gesetzt, und der Blitter hat absolute Priorität über den 68000.

Erklärungen:

Wie man sieht, kommt es vor, daß die Ausgangsdaten D0 erst ins RAM gelangen, nachdem schon die A1, B1 und C1 Daten gelesen wurden. Dies kommt von dem sogenannten "Pipelining" innerhalb des Blitters. Pipelining bedeutet, daß die Verarbeitung der Daten innerhalb des Blitters in mehreren Stufen abläuft, die voneinander unabhängig arbeiten. Jede Stufe ist mit dem

Ausgang der vorangehenden und dem Eingang der nachfolgenden verbunden. Die erste Stufe bekommt die Eingangsdaten (z.B. A0, B0 und C0), verarbeitet sie und gibt sie an die zweite weiter. Während sie in dieser Stufe weiterverarbeitet werden, gelangen schon die nächsten Eingangsdaten in die Eingangsstufe (A1, B1 und C1). Wenn die ersten Daten dann zur Ausgangsstufe gelangen (D0), hat der Blitter schon längst die nächsten Daten gelesen. Es befinden sich während einer Blitter-Operation zu jedem Zeitpunkt immer zwei Datenpaare in unterschiedlichen Verarbeitungsstufen im Blitter.

Die Tabelle ermöglicht auch die Berechnung der Ablaufdauer einer Blitter-Operation. In jeder Mikrosekunde hat der Blitter zwei Buszyklen zur Verfügung. Soll z.B. ein Bereich von 64 Kilobyte (32768 Worte) von A nach D kopiert werden, benötigt der Blitter $2 \cdot 32768$ Zyklen. Wenn derselbe Bereich aber noch mit Quelle C kombiniert werden soll, ergeben sich $3 \cdot 32768$ Zyklen, da für jedes Wort noch ein Datenwort aus Quelle C gelesen werden muß.

Die Tabelle zeigt auch, daß der Blitter nicht in der Lage ist, jeden Buszyklus zu nutzen, wenn nur ein Blitter-DMA-Kanal aktiv ist.

Beispielprogramme

Programm 1: Linien mit dem Blitter

Dieses Programm stellt eine universelle Routine zum Zeichnen von Linien mit dem Blitter zur Verfügung. Sie zeigt, wie man die notwendigen Werte berechnen kann. Ansonsten ist das Programm einfach:

Im Vorprogramm wird der Speicher angefordert und die Copper-List aufgebaut. Unbekannt dürfte nur die OwnBlitter-Routine sein. Wie ihr Name schon sagt, kann man mit ihr den Blitter für sich beanspruchen. Entsprechend steht daher am Ende des Programms das Gegenstück dazu, der DisownBlitter-Aufruf,

mit dem der Blitter wieder unter die Kontrolle des Betriebssystems gestellt wird.

Das Programm verwendet nur eine einzige Hires-Bitplane, mit den Standardmaßen 640 auf 256 Punkte. In der Hauptschleife zeichnet das Programm Linien, die immer von einer Bildschirmkante durch den Bildmittelpunkt zur gegenüberliegenden Seite gehen. Immer, wenn ein Bildschirm auf diese Weise gefüllt wurde, verschiebt das Programm die Maske, mit der die Linien gezeichnet werden, und beginnt wieder von vorne.

Anmerkung: Die Koordinatenangaben in dem Programm gehen von einem Punkt 0,0 in der oberen, linken Bildecke aus, sind also keine mathematischen Koordinaten, wie sie in den vorangegangenen Erklärungen benutzt wurden. Praktisch hat das die Auswirkung, daß in Vergleichen der Y-Werte das Größer/Kleiner-Zeichen umgedreht werden muß.

;*** Linien mit dem Blitter

;Customchipregister

INTENA = \$9A ;Interruptenable-Register (schreiben)
DMACON = \$96 ;DMA-Controllregister (schreiben)
DMACONR = \$2 ;DMA-Controllregister (lesen)
COLOR00 = \$180 ;Farbpalettenregister 0
VHPOSR = \$6 ;Strahlposition (lesen)

;Copper Register

COP1LC = \$80 ;Adresse der 1. Copper-List
COP2LC = \$84 ;Adresse der 2. Copper-List
COPJMP1 = \$88 ;Sprung nach Copper-List 1
COPJMP2 = \$8a ;Sprung nach Copper-List 2

;Bitplane Register

BPLCON0 = \$100 ;Bitplane Kontrollregister 0
BPLCON1 = \$102 ;1 (Scrollwerte)
BPLCON2 = \$104 ;2 (Sprite<>Playfield Priorität)
BPL1PTH = \$0E0 ;Zeiger auf 1. Bitplane
BPL1PTL = \$0E2 ;
BPL1MOD = \$108 ;Modulo-Wert für ungerade Bit-Planes

```

BPL2MOD = $10A ;Module-Wert für gerade Bit-Planes
DIWSTRT = $08E ;Start des Bildschirmfensters
DIWSTOP = $090 ;Ende des Bildschirmfensters
DDFSTRT = $092 ;Bit-Plane DMA Start
DDFSTOP = $094 ;Bit-Plane DMA Stop

```

```

;Blitter Register

```

```

BLTCON0 = $40 ;Blitter-Kontrollregister 0 (ShiftA,Usex,LFx)
BLTCON1 = $42 ;Blitter-Kontrollregister 1 (ShiftB,ver. Bits)
BLTCPTH = $48 ;Zeiger auf Quelle C
BLTCPTL = $4a
BLTBPTH = $4c ;Zeiger auf Quelle B
BLTBPTL = $4e
BLTAPTH = $50 ;Zeiger auf Quelle A
BLTAPTL = $52
BLTDPTH = $54 ;Zeiger auf Zieldaten D
BLTDPTL = $56
BLTCMOD = $60 ;Modulo-Wert für Quelle C
BLTBMOD = $62 ;Modulo-Wert für Quelle B
BLTAMOD = $64 ;Modulo-Wert für Quelle A
BLTDMOD = $66 ;Modulo-Wert für Ziel D
BLTSIZE = $58 ;Höhe und Breite des Blitter-Fensters
BLTCDAT = $70 ;Datenregister Quelle C
BLTBDAT = $72 ;Datenregister Quelle B
BLTADAT = $74 ;Datenregister Quelle A
BLTAFWM = $44 ;Maske für das erste Datenwort von Quelle A
BLTALWM = $46 ;Maske für das erste Datenwort von Quelle B

```

```

;CIA-A Portregister A (Maustaste)

```

```

CIAAPRA = $bfe001

```

```

;Exec Library Base Offsets

```

```

OpenLibrary = -30-522 ;LibName,Version/a1,d0
Forbid = -30-102
Permit = -30-108
AllocMem = -30-168 ;ByteSize,Requirements/d0,d1
FreeMem = -30-180 ;MemoryBlock,ByteSize/a1,d0

```

```

;Graphics Library Base Offsets

```

```

OwnBlitter = -30-426
DisownBlitter = -30-432

```

```
;graphics base

StartList = 38

;sonstige Labels

Execbase = 4
Planesize = 80*256 ;Größe der Bitplane: 80 Bytes auf 256 Zeilen
Planewidth = 80
CLsize = 3*4 ;Die Copper-List enthält 3 Befehle
Chip = 2 ;Chip-RAM anfordern
Clear = Chip+$10000 ;Chip-RAM vorher löschen

;*** Vorprogramm ***

Start:

;Speicher für die Bitplane anfordern

move.l Execbase,a6
move.l #Planesize,d0 ;Speicherbedarf der Plane
move.l #clear,d1
jsr AllocMem(a6) ;Speicher anfordern
move.l d0,Planeadr
beq Ende ;Fehler! -> Ende

;Speicher für Copper-List anfordern

moveq #CLsize,d0
moveq #chip,d1
jsr AllocMem(a6)
move.l d0,CLadr
beq FreePlane ;Fehler! -> FreePlane

;Copper-List erstellen

move.l d0,a0 ;Adresse der Copper-List nach a0
move.l Planeadr,d0 ;Adresse der Bitplane
move.w #bpl1pth,(a0)+ ;ersten Copperbefehl ins RAM
swap d0
move.w d0,(a0)+ ;Hi-Wort der Bitplaneadresse ins RAM
move.w #bpl1ptl,(a0)+ ;zweiten Befehl ins RAM
swap d0
move.w d0,(a0)+ ;Lo-Wort der Bitplaneadresse ins RAM
move.l #$fffffffe,(a0) ;Ende der Copper-List
```

;Blitter anfordern

```
move.l #GRname,a1
clr.l d0
jsr OpenLibrary(a6)
move.l a6,-(sp) ;ExecBase auf den Stack
move.l d0,a6 ;GraphicsBase nach a6
move.l a6,-(sp) ;und auch auf den Stack
jsr OwnBlitter(a6) ;Blitter übernehmen
```

;*** Hauptprogramm ***

;DMA und Task-Switching sperren

```
move.l 4(sp),a6 ;ExecBase nach a6
jsr forbid(a6) ;Task-Switching aus
lea $dff000,a5
move.w #$03e0,dmacon(a5)
```

;Copper initialisieren

```
move.l CLAdr,cop1lc(a5)
clr.w copjmp1(a5)
```

;Farben festlegen

```
move.w #$0000,color00(a5) ;Hintergrund schwarz
move.w #$0fa0,color00+2(a5) ;Linien gelb
```

;Playfield initialisieren

```
move.w #$3081,diwstrt(a5) ;30,129
move.w #$30c1,diwstop(a5) ;30,449
move.w #$003c,ddfstrt(a5) ;normaler Hires-Bildschirm
move.w #$00d4,ddfstop(a5)
move.w %#1001001000000000,bplcon0(a5)
clr.w bplcon1(a5)
clr.w bplcon2(a5)
clr.w bpl1mod(a5)
clr.w bpl2mod(a5)
```

;DMA ein

```
move.w #$83C0,dmacon(a5)
```

;Linien zeichnen

;Startwerte festlegen:

```
move.l Planeadr,a0 ;konstante Parameter für DrawLine
move.w #Planewidth,a1 ;in die entsprechenden Register
move.w #255,a3 ;Maße der Bitplane in Register
move.w #639,a4
move.w #$0303,d7 ;Startmuster
```

```
Loop: rol.w #2,d7 ;Muster verschieben
move.w d7,a2 ;Muster in Register für DrawLine
```

```
clr.w d6 ;Schleifenvariable löschen
SchleifeX:
clr.w d1 ;Y1 = 0
move.w a3,d3 ;Y2 = 255
move.w d6,d0 ;X1 = Schleifenvariable
move.w a4,d2
sub.w d6,d2 ;X2 = 639-Schleifenvariable
```

bsr DrawLine ;Linie zeichnen

```
addq.w #4,d6 ;Schleifenvariable erhöhen
cmp.w a4,d6 ;Test ob schon größer als 639
ble.s SchleifeX ;wenn nein, mit der Schleife weitermachen
```

```
clr.w d6 ;Schleifenvariable löschen
SchleifeY:
move.w a4,d0 ;X1 = 639
clr.w d2 ;X2 = 0
move.w d6,d1 ;Y1 = Schleifenvariable
move.w a3,d3
sub.w d6,d3 ;Y2 = 255-Schleifenvariable
```

bsr DrawLine ;Linie zeichnen

```
addq.w #2,d6 ;Schleifenvariable erhöhen
cmp.w a3,d6 ;Test ob schon größer als 255
ble.s SchleifeY ;wenn nein, mit der Schleife weitermachen
```

```
btst #6,ciaapra ;Maustaste gedrückt?
bne Loop ;Nein -> weitermachen
```

```
;*** Nachprogramm ***
```

```
;Warten, bis Blitter fertig ist
```

```
Wait: btst #14,dmaconr(a5)
```

```
bne Wait
```

```
;Alte Copper-List wieder aktivieren
```

```
move.l (sp)+,a6 ;GraphicsBase vom Stack holen
```

```
move.l StartList(a6),cop1lc(a5)
```

```
clr.w copjmp1(a5) ;Startup-Copper-List aktivieren
```

```
move.w #$8020,dmacon(a5)
```

```
jsr DisownBlitter(a6) ;Blitter freigeben
```

```
move.l (sp)+,a6 ;ExecBase vom Stack
```

```
jsr Permit(a6) ;TaskSwitching wieder erlauben
```

```
;Speicher für Copper-List wieder freigeben
```

```
move.l CLadr,a1 ;Parameter für FreeMem setzen
```

```
moveq #CLsize,d0
```

```
jsr FreeMem(a6) ;Speicher freigeben
```

```
;Speicher für Bitplane wieder freigeben
```

```
FreePlane:
```

```
move.l Planeadr,a1
```

```
move.l #Planesize,d0
```

```
jsr FreeMem(a6)
```

```
Ende:
```

```
clr.l d0
```

```
rts ;Programm verlassen
```

```
;Variablen
```

```
CLadr: dc.l 0 ;Adresse der Copper-List
```

```
Planeadr: dc.l 0 ;Adresse der Bitplane
```

```
;Konstanten
```

```
GRname: dc.b "graphics.library",0
```

```
even
```

```

;*** DrawLine Routine ***
;DrawLine zeichnet eine Linie mit dem Blitter.
;Folgende Parameter werden benötigt:
;d0 = X1  X-Koordinate des Startpunkts
;d1 = Y1  Y-Koordinate des Startpunkts
;d2 = X2  X-Koordinate des Endpunkts
;d3 = Y2  Y-Koordinate des Endpunkts
;a0 muss auf das erste Wort der Bitplane zeigen
;a1 enthält die Breite der Bitplane in Bytes
;a2 wird direkt in das Maskenregister geschrieben
;d4 bis d6 werden als Arbeitsregister verwendet

```

DrawLine:

;Berechnung der Anfangsadresse der Linie

```

move.l a1,d4 ;Breite in Arbeitsregister
mulu d1,d4   ;Y1 * Bytes pro Zeile
moveq #-10,d5 ;Vorzeichenlos: $f0
and.w d0,d5 ;untere vier Bits von X1 ausmaskieren
lsr.w #3,d5 ;Rest durch 8 teilen
add.w d5,d4 ;Y1 * Bytes pro Zeile + X1/8
add.l a0,d4 ;plus Anfangsadresse der Bitplane
;d4 enthält jetzt die Startadresse
;der Linie
;Oktant und Deltas errechnen

clr.l d5 ;Arbeitsregister löschen
sub.w d1,d3 ;Y2-Y1 DeltaY nach D3
roxl.b #1,d5 ;Vorzeichen von DeltaY in d5 schieben
tst.w d3 ;N-Flag restaurieren
bge.s y2gy1 ;Wenn DeltaY positiv, nach y2gy1
neg.w d3 ;DeltaY invertieren (es wird dadurch positiv)
y2gy1:
sub.w d0,d2 ;X2-X1 DeltaX nach D2
roxl.b #1,d5 ;Vorzeichen von DeltaX in d5 schieben
tst.w d2 ;N-Flag restaurieren
bge.s x2gx1 ;Wenn DeltaX positiv, nach x2gx1
neg.w d2 ;DeltaX invertieren (es wird dadurch positiv)
x2gx1:
move.w d3,d1 ;DeltaY nach d1
sub.w d2,d1 ;DeltaY-DeltaX
bge.s dygdx ;Wenn DeltaY > DeltaX, nach dygdx
exg d2,d3 ;kleineres Delta nach d2
dygdx: roxl.b #1,d5 ;d5 enthält das Ergebnis der 3 Vergleiche
move.b Oktabelle(pc,d5),d5 ;zugehörigen Oktanten holen

```

```
add.w d2,d2 ;kleines Delta * 2
```

```
;Test, ob der Blitter die letzte Operation schon beendet hat
```

```
WBlit: btst #14,dmaconr(a5) ;BBUSY-Bit testen
```

```
bne.s WBlit ;Warten, bis es gleich 0 ist
```

```
move.w d2,bltbmod(a5) ;2* kleines Delta nach BLTBMOD
```

```
sub.w d3,d2 ;2* kleines Delta - grosses Delta
```

```
bge.s signnl ;Wenn 2*Kdelta > Gdelta nach signnl
```

```
or.b #$40,d5 ;Signflag setzen
```

```
signnl: move.w d2,bltaptl(a5) ;2*Kdelta-Gdelta in BLTAPTL
```

```
sub.w d3,d2 ;2* kleines Delta - 2* grosses Delta
```

```
move.w d2,bltamod(a5) ;nach BLTAMOD
```

```
;restliche Register initialisieren
```

```
move.w #$8000,bltadat(a5)
```

```
move.w a2,bltbdat(a5) ;Maske aus a2 in BLTBDAT
```

```
move.w #$ffff,bltafwm(a5)
```

```
and.w #$000f,d0 ;untere 4 Bits von X1
```

```
ror.w #4,d0 ;an START0-3 schieben
```

```
or.w #$0bca,d0 ;USEx und Lfx setzen
```

```
move.w d0,bltcon0(a5)
```

```
move.w d5,bltcon1(a5) ;Oktant in Blitter
```

```
move.l d4,bltcpth(a5) ;Startadresse der Linie nach
```

```
move.l d4,bltdpth(a5) ;BLTCPT und BLTDPT
```

```
move.w a1,bltcmmod(a5) ;Breite der Bitplane in die
```

```
move.w a1,bltdmod(a5) ;beiden Modulo-Register
```

```
;BLTSIZE initialisieren und Blitter starten
```

```
lsl.w #6,d3 ;Länge * 64
```

```
addq.w #2,d3 ;plus (Width = 2)
```

```
move.w d3,bltsize(a5)
```

```
rts
```

```
;Oktantentabelle mit LINE =1:
```

```
;Die Oktantentabelle enthält für jeden Oktanten den zuge-
```

```
;hörigen Code-Wert, der schon an die richtige Position ge-
```

```
;schoben ist. Außerdem ist noch das LINE-Bit gesetzt.
```

Okttabelle:

```
dc.b 0 *4+1 ;y1<y2, x1<x2, dx<dy = 0kt6
dc.b 4 *4+1 ;y1<y2, x1<x2, dx>dy = 0kt7
dc.b 2 *4+1 ;y1<y2, x1>x2, dx<dy = 0kt5
dc.b 5 *4+1 ;y1<y2, x1>x2, dx>dy = 0kt4
dc.b 1 *4+1 ;y1>y2, x1<x2, dx<dy = 0kt1
dc.b 6 *4+1 ;y1>y2, x1<x2, dx>dy = 0kt0
dc.b 3 *4+1 ;y1>y2, x1>x2, dx<dy = 0kt2
dc.b 7 *4+1 ;y1>y2, x1>x2, dx>dy = 0kt3
```

Programm 2: Flächen füllen mit dem Blitter

Dieses Programm ähnelt sehr dem ersten Programm. Es zeigt, wie man durch Zeichnen von Begrenzungslinien und Füllen mit dem Blitter ausgemalte Vielecke erzeugen kann.

Da es größtenteils mit dem ersten Programm identisch ist, haben wir hier lediglich die Teile abgedruckt, die in Programm 1 ausgetauscht werden müssen, um Programm 2 zu erhalten.

Dies ist einmal der Abschnitt von dem Kommentar "Linien zeichnen" bis zum Kommentar "*** Nachprogramm ***". Dieser Bereich muß gegen den nachfolgenden Abschnitt mit der Bezeichnung "Teil 1" ausgetauscht werden.

Außerdem muß die alte Oktantentabelle am Ende des Programms durch die neue nach der Überschrift "Teil 2" ersetzt werden.

Die neue Oktantentabelle ist notwendig, da der Blitter zum Füllen der Flächen ja Begrenzungslinien mit nur einem Punkt pro Zeile benötigt. In der neuen Oktantentabelle ist daher zusätzlich zum LINE-Bit auch noch das SING-Bit gesetzt.

Das mit "Teil 1" bezeichnete Programm zieht zwei Linien und läßt den dazwischenliegenden Bereich vom Blitter auffüllen. Danach wartet es auf die Maustaste.

```
*** Flächen füllen mit dem Blitter ***
```

```
Teil 1:
```

```
;ausgefülltes Dreieck zeichnen
```

```
;Startwerte festlegen
```

```
move.l Planeadr,a0 ;konstante Parameter für
move.w #Planewidth,a1 ;die LineDraw-Routine setzen
move.w #$ffff,a2 ;Maske auf $FFFF -> kein Muster
```

```
* Begrenzungslinien zeichnen *
```

```
;Linie von 320,10 nach 600,230
move.w #320,d0
move.w #10,d1
move.w #600,d2
move.w #230,d3
bsr.l drawline ;Linie zeichnen
```

```
;Linie von 319,10 nach 40,230
move.w #319,d0
move.w #10,d1
move.w #40,d2
move.w #230,d3
bsr.l drawline ;Linie zeichnen
```

```
* Fläche auffüllen *
```

```
;Warten, bis Blitter die letzte Linie gezeichnet hat
```

```
Wline: btst #14,dmaconr(a5) ;BBUSY teste
bne.s Wline
```

```
add.l #Planesize-2,a0 ;Adresse des letzten Worts
move.w #$09f0,bltcon0(a5) ;USEA und D, Lfx: D = A
move.w #$000a,bltcon1(a5) ;Inclusive Fill plus Descending
move.w #$ffff,bltafwm(a5) ;First- und Lastwordmask setzen
move.w #$ffff,bltalwm(a5)
move.l a0,bltaph(a5) ;Adresse des letzten Worts der Bit-
move.l a0,bltdpth(a5) ;plane in die Adreßzeiger-Register
move.w #0,bltamod(a5) ;kein Modulo
move.w #0,bltdmod(a5)
move.w #$ff*64+40,bltsize(a5) ;Blitter starten
```

;auf Maustaste warten

end: btst #6,ciaapra ;Maustaste gedrückt?
bne.S end ;Nein -> weitermachen

Ende von Teil 1.

Teil 2:

;Okttantentabelle mit SING =1 und LINE =1:

Okttabelle:

dc.b 0 *4+3 ;y1<y2, x1<x2, dx<dy = 0kt6
dc.b 4 *4+3 ;y1<y2, x1<x2, dx>dy = 0kt7
dc.b 2 *4+3 ;y1<y2, x1>x2, dx<dy = 0kt5
dc.b 5 *4+3 ;y1<y2, x1>x2, dx>dy = 0kt4
dc.b 1 *4+3 ;y1>y2, x1<x2, dx<dy = 0kt1
dc.b 6 *4+3 ;y1>y2, x1<x2, dx>dy = 0kt0
dc.b 3 *4+3 ;y1>y2, x1>x2, dx<dy = 0kt2
dc.b 7 *4+3 ;y1>y2, x1>x2, dx>dy = 0kt3

1.5.8 Die Tonausgabe

Grundlagen elektronischer Musik

Wenn wir etwas hören, egal ob Musik, Geräusche oder Sprache, geschieht dies in Form von Schwingungen der Luft, den Schallwellen, die unser Ohr erreichen. Ein normales Musikinstrument erzeugt diese Schwingungen entweder direkt, wie z.B. eine Flöte, bei ihr wird die Luft durch Hineinblasen in Schwingungen versetzt, oder indirekt, indem erst ein Teil des Instruments den Ton erzeugt und ihn dann an die Luft weitergibt. Dies geschieht beispielsweise bei allen Saiteninstrumenten.

Ein elektronisches Instrument erzeugt in seinen Schaltkreisen elektrische Schwingungen, die in ihrem Verlauf dem gewünschten Ton entsprechen. Hörbar werden sie erst, wenn sie mittels eines Lautsprechers in Schallschwingungen umgewandelt werden. Beim Amiga verwendet man dazu meistens den im Monitor eingebauten Lautsprecher. Leider ist dieser aufgrund seiner Größe

und Qualität nicht in der Lage, die elektrischen Schwingungen in identische Schallwellen umzusetzen. Auf deutsch gesagt: Er klingt schlecht. Man sollte daher den Amiga unbedingt an eine gute Verstärkeranlage anschließen, um in den vollen Genuß seiner musikalischen Fähigkeiten zu kommen. Doch zurück zum Thema:

Welche Parameter bestimmen den Ton, der aus dem Computer kommt?

Frequenz

Als erstes ist da die Frequenz eines Tons. Sie bestimmt, ob er hoch oder tief klingt. Physikalisch gesehen ist die Frequenz die Anzahl der Schwingungen in der Sekunde, gemessen wird sie in Hertz (Hz). 1 Schwingung pro Sekunde ist 1 Hz, folgerichtig entspricht ein Kilohertz dann 1000 Hertz. Das menschliche Ohr kann Töne zwischen 16 und 16000 Hertz wahrnehmen. Wer etwas von Musik versteht, weiß, daß das eingestrichene "a" eine Frequenz von 440 Hz hat. Der Zusammenhang zwischen Frequenz und Tonhöhe ist wie folgt: Mit jeder Oktave verdoppelt sich die Frequenz. Das zweigestrichene "a" hat demzufolge eine Frequenz von 880 Hz, während das kleine "a" (eine Oktave unter dem eingestrichenen "a") mit einer Frequenz von 220 Hz erklingt.

Die Frequenz eines Tones muß nicht unbedingt konstant sein. Sie kann z.B. periodisch einige Hz um die eigentliche Tonhöhe schwanken. Dieser Effekt heißt Vibrato.

Lautstärke

Der zweite Parameter eines Tones ist seine Lautstärke. Unter der Lautstärke versteht man, vereinfacht gesagt, die Amplitude einer Schwingung. Gemessen wird die Lautstärke eines Tones in Dezibel (dB). Der Bereich menschlichen Hörens reicht von 1 dB bis ca. 120 dB. Etwa 10 dB bewirken eine Verdoppelung der Lautstärke. Die Lautstärke des Schalls bezeichnet man auch als Schalldruck.

Die Lautstärke kann durch viele Parameter beeinflusst werden. Am einfachsten natürlich durch den Lautstärkeregler am Monitor bzw. Verstärker. Dieser macht nichts anderes, als die Amplitude der elektrischen Schwingung zu verändern. Aber auch der Abstand zwischen Zuhörer und Lautsprecher hat eine Auswirkung auf die Lautstärke. Je weiter man sich vom Lautsprecher entfernt, desto leiser wird der Ton. Aber auch die Raumeinrichtung, offene oder geschlossene Türen usw., all dies kann sich auf die Amplitude der Schallwellen auswirken. Aus diesem Grund ist die absolute Lautstärke auch nicht so wichtig. Entscheidend ist vielmehr die relative Lautstärke der Töne untereinander, z.B. ob der nachfolgende Ton lauter oder leiser als sein Vorgänger ist.

Zwischen der Lautstärke eines Tones und seiner Frequenz besteht ein Zusammenhang. Der Grund dafür ist die Empfindlichkeit des menschlichen Ohres. Hohe und tiefe Töne werden leiser empfunden als die mittleren, auch wenn sie physikalisch denselben Schalldruck in Dezibel haben. Dieser mittlere Tonhöhenbereich reicht ganz grob von etwa 1000 bis 3000 Hz. Innerhalb dieses Frequenzbereichs befinden sich die Schwingungen der menschlichen Sprache, was vielleicht der Grund für die dort höhere Empfindlichkeit ist.

Auch die Lautstärke eines Tones kann sich innerhalb eines gewissen Rahmens periodisch verändern, diesen Effekt nennt man Tremolo. Dazu kommt noch der Lautstärkeverlauf vom Beginn bis zum Ende des Tones. Ein Ton kann laut beginnen und dann langsam ausklingen. Er kann aber auch laut beginnen, dann auf ein bestimmtes Maß absinken und zum Schluß abrupt abbrechen. Oder er beginnt leise und steigert dann langsam seine Lautstärke. Es gibt hier fast unendlich viele Kombinationsmöglichkeiten.

Klangfarbe

typische Wellenformen

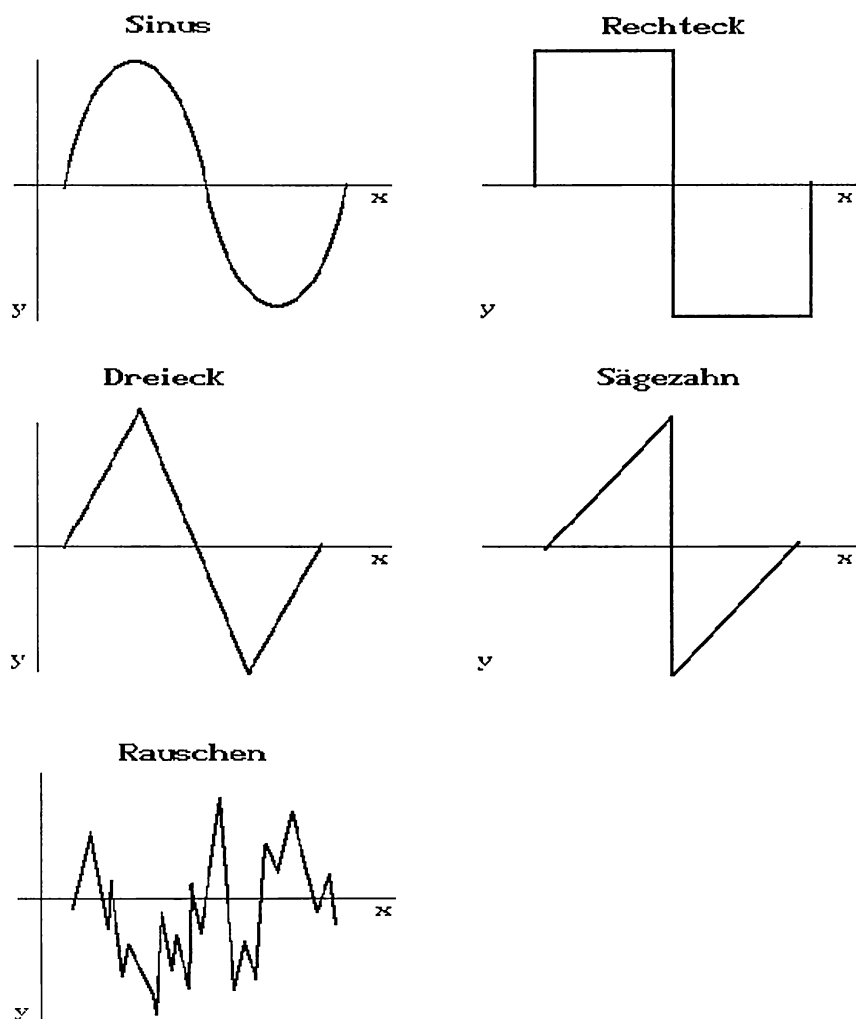


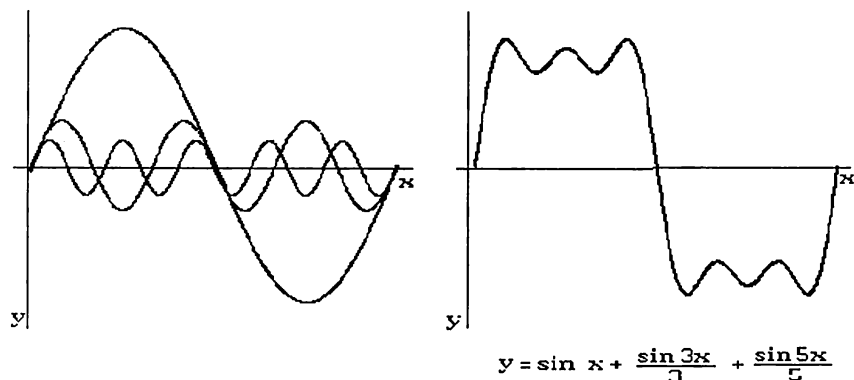
Abb. 1.5.8.1

Der dritte und letzte Parameter eines Tones ist etwas komplizierter. Es handelt sich dabei um die Klangfarbe. Sie spielt eine wichtige Rolle. Es gibt hunderte verschiedener Instrumente, die alle einen Ton mit gleicher Frequenz und Lautstärke spielen können, aber trotzdem klingt jedes davon anders. Der Grund dafür liegt in der Form der Schwingung. Die Abbildung 1.5.8.1 zeigt vier häufig vorkommende Wellenformen. Warum klingen sie unterschiedlich?

Jede Wellenform, egal wie sie auch aussieht, besteht immer aus einem Gemisch von Sinusschwingungen, die untereinander in einem festen Frequenzverhältnis stehen. Bei einem Rechteck z.B. hat der erste Teilton die Grundfrequenz des Tons, der zweite hat die dreifache Frequenz, aber nur noch ein Drittel der Amplitude. Der dritte Teilton hat die fünffache Frequenz und ein Fünftel der Amplitude usw.

Zusammensetzung verschiedener Wellenformen aus Sinusschwingungen

Schon 3 Sinuskurven ergeben ein brauchbares Rechteck



oder einen Sägezahn

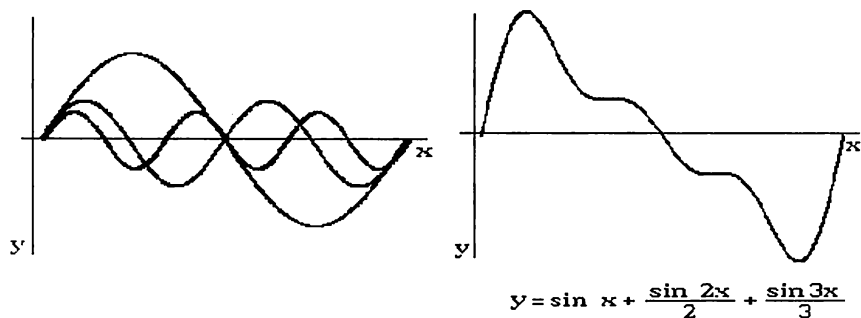


Abb. 1.5.8.2

Abbildung 1.5.8.2 zeigt dies sowohl für eine Rechteckschwingung als auch für einen Sägezahn. Der Einfachheit halber sind jeweils nur die ersten drei Teiltöne angegeben.

Wie gesagt, setzen sich alle periodischen Wellenformen aus Sinusschwingungen zusammen. Man nennt sie die Obertonreihe eines Tons. Die reine Sinusschwingung besteht logischerweise nur aus dem 1. Teilton. Eine Rechteckschwingung enthält unendlich viele Obertöne. Die Anzahl der Obertöne und ihr Frequenz- und Amplitudenverhältnis untereinander bestimmen allein über die Klangfarbe eines Tones. Die Obertonreihe ist deshalb so wichtig, weil das menschliche Ohr nur auf Sinusschwingungen reagiert. Ein Ton, dessen Wellenform vom reinen Sinus abweicht, wird schon im Ohr, also noch vor dem eigentlichen Hören, in seine Teiltöne zerlegt. Man sollte diese Tatsachen während den folgenden Erläuterungen im Hinterkopf behalten.

Geräusche

Außer Tönen gibt es auch noch Geräusche. Während man einen Ton sehr genau definieren und auch elektronisch erzeugen kann, ist dies bei den Geräuschen sehr viel schwieriger. Sie besitzen weder eine bestimmte Frequenz noch einen definierten Lautstärkeverlauf und auch keine einheitliche Wellenform. Sie stellen eine willkürliche Kombination von Schall-Ereignissen dar. Die Grundlage vieler Geräusche ist daher das Rauschen, denn Rauschen ist eine Mischung unendlich vieler Schwingungen, deren Frequenzen und Phasenlagen in keinem bestimmten Verhältnis zueinander stehen. Der Wind rauscht beispielsweise, weil jedes der abermillionen Luftmoleküle beim Zusammenstoß untereinander oder mit den Gegenständen auf dem Erdboden in Eigenschwingungen versetzt wird, die alle zusammen ein undefinierbares Klanggemisch bilden, eben das typische Windrauschen.

Die Tonerzeugung beim Amiga

Digitalisierung einer Wellenform

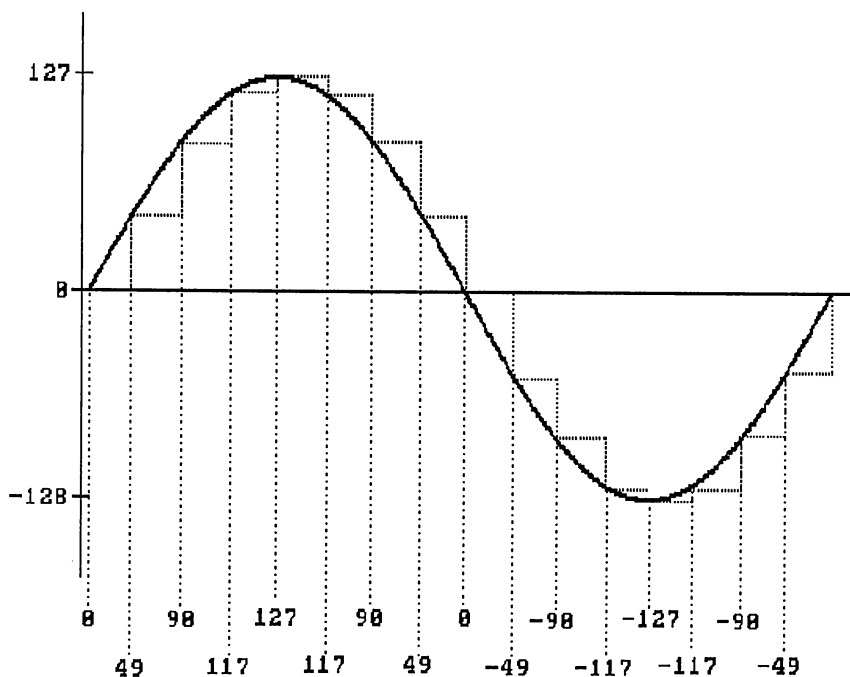


Abb. 1.5.8.3

Das Hauptkriterium für die Beurteilung der akustischen Fähigkeiten eines Computers ist seine Vielseitigkeit. Optimal wäre es, wenn sich alle drei Parameter eines Tons, Frequenz, Lautstärke und Klangfarbe, völlig frei einstellen ließen.

Beim Amiga hat man versucht, sich diesem Ziel möglichst weit zu nähern. Um nicht an vorgegebene Wellenformen gebunden zu sein, wird das digitale Äquivalent der gewünschten Wellenform im Speicher abgelegt und dann mittels eines Digital/Analog-

Wandler in die entsprechende elektrische Schwingung umgewandelt. Anders ausgedrückt, die Schwingung wird digitalisiert und im Computer gespeichert. Bei der Ausgabe werden die digitalisierten Daten wieder analogisiert und an den Verstärker ausgegeben.

In der Abbildung 1.5.8.1 sieht man unterschiedliche Wellenformen. Wenn diese in eine für den Computer verständliche Form umgewandelt werden sollen, muß ihr Verlauf durch Zahlen dargestellt werden.

Dazu teilt man eine Schwingung der gewünschten Wellenform in eine gerade Anzahl gleich großer Abschnitte ein. Man beginnt damit möglichst beim Nulldurchgang der Kurve. In jedem dieser Abschnitte überträgt man den zugehörigen Y-Wert in den Speicher. Man hat danach eine Zahlenfolge, deren Elemente quasi Momentaufnahmen der Schwingung zu bestimmten Zeitpunkten darstellen. Der englische Begriff für diese digitalisierten Werte lautet "sample", was auf deutsch soviel wie "Probe" heißt.

Bei der Ausgabe wandelt der Amiga die Zahlenwerte aus dem Speicher wieder in die entsprechenden Ausgangsspannungen um. Da die Schwingung bei der Digitalisierung aber nur in eine begrenzte Anzahl Samples zerlegt wurde, kann die Ausgangskurve auch nur mit dieser Anzahl rekonstruiert werden. Dadurch entsteht der treppenförmige Verlauf dieser Schwingung, den man in Abbildung 1.5.8.3 sehen kann.

Die Qualität der so reproduzierten Klänge im Vergleich zu ihren Ursprungsschwingungen ist im wesentlichen von zwei Größen abhängig:

Die eine ist die Auflösung der digitalisierten Signale. Damit ist der Wertebereich eines Samples gemeint. Beim Amiga beträgt er acht Bit, reicht also von -128 bis +127. Jeder Eingangswert kann einen von 256 Werten im Speicher annehmen. Da die Auflösung des analogen Eingangssignals theoretisch unendlich, die der einzelnen Samples aber begrenzt ist, entstehen hier Fehler. Man nennt sie Quantisierungs- oder Rundungsfehler. Denn wenn der Eingangswert irgendwo zwischen zwei Zahlen liegt, also nicht

genau einer der 256 Digitalisierungsstufen entspricht, wird er auf- oder abgerundet. Der maximal mögliche Quantisierungsfehler beträgt $1/256$ des digitalisierten Wertes (man sagt auch: der Fehler beträgt 1 LSB).

Mit dem Quantisierungsfehler ist das sogenannte Quantisierungsrauschen verbunden. Wie der Name schon sagt, äußert es sich als Rauschen, das mit der Größe des Quantisierungsfehlers zunimmt.

Ein Wertebereich von acht Bit erlaubt schon eine sehr gute Reproduktion der Ursprungsschwingung. Für Hifi-Qualität benötigt man aber eine höhere Auflösung. Ein CD-Plattenspieler arbeitet z.B. mit 16 Bit.

Der zweite Parameter für die Qualität digitalisierter Klänge ist die sogenannte Sampling-Rate. Damit ist die Anzahl der Samples pro Sekunde gemeint. Eine höhere Anzahl Samples bringt natürlich auch eine bessere Wiedergabe mit sich. Die Sampling-Rate läßt sich beim Amiga innerhalb gewisser Grenzen frei einstellen. Vorher muß man sich aber überlegen, wie viele Samples man pro digitalisierter Schwingung verwenden will. In unserem Beispiel in Abbildung 1.5.8.3 sind es 16 Werte. Zwischen der daraus resultierenden treppenförmigen Sinusschwingung und einem normalen Sinussignal kann man kaum mehr einen hörbaren Unterschied feststellen.

Die Ausgabe der digitalisierten Töne

Hat man die gewünschte Wellenform in die entsprechenden Zahlen umgewandelt und in den Speicher geschrieben, will man sie natürlich zum Erklingen bringen. Der Amiga besitzt vier Tonkanäle, die alle nach dem gleichen Prinzip arbeiten:

Eine digitalisierte Schwingung wird per DMA aus dem Speicher gelesen und mittels eines Digital/Analog-Wandlers ausgegeben. Dieser Vorgang wiederholt sich kontinuierlich, so daß aus der einzelnen Schwingung ein dauerhafter Ton wird.

Die Kanäle 0 und 3 werden zum linken, 1 und 2 zum rechten Stereokanal zusammengefaßt.

Jeder Audiokanal besitzt einen zugehörigen DMA-Kanal. Da der DMA-Zugriff beim Amiga immer wortweise abläuft, faßt man zwei Samples zu einem Datenwort zusammen. Aus diesem Grund benötigt man immer eine gerade Anzahl Samples. Die obere Hälfte des Worts (Bits 8-15) wird immer vor der unteren (Bits 0-7) ausgegeben. Die Datenliste für unsere digitalisierte Sinus-schwingung sieht dann im Speicher wie folgt aus, wobei "Start" die Anfangsadresse der Liste im Chip-RAM ist:

Start:

dc.b 0,49 ;1. Datenwort, Samples 1 und 2
dc.b 90,117 ;2. Datenwort, Samples 3 und 4
dc.b 127,117 ;3. Datenwort, Samples 5 und 6
dc.b 90,49 ;4. Datenwort, Samples 7 und 8
dc.b 0,-49 ;5. Datenwort, Samples 9 und 10
dc.b -90,-117 ;6. Datenwort, Samples 11 und 12
dc.b -127,-117;7. Datenwort, Samples 13 und 14
dc.b -90,-49 ;8. Datenwort, Samples 15 und 16

End:

Der Digital/Analog-Wandler benötigt die Samples als vorzeichenbehaftete 8-Bit-Zahlen. Wie in der gesamten Digitaltechnik üblich, müssen sie in Form eines Zweierkomplements angegeben werden. Diese Umrechnung erledigt für uns der Assembler, so daß man die negativen Werte direkt in die Datenliste schreiben kann.

Jetzt muß man einen der vier Audiokanäle wählen, über den man den Ton ausgeben will. Danach wird der zugehörige Audio-DMA-Kanal initialisiert. Fünf Register pro Kanal legen die Betriebsparameter fest. Die ersten beiden bilden ein Adreßregisterpaar, wie man es schon von anderen DMA-Kanälen kennt. Sie heißen AUDxLCH und AUDxLCL, oder gemeinsam AUDxLC, wobei x der Nummer des DMA-Kanals entspricht:

Reg.	Name	Funktion
\$0A0	AUD0LCH	Zeiger auf die Audiodaten Bits 16-18 von Kanal 0 Bits 0-15
\$0A2	AUD0LCL	
\$0B0	AUD1LCH	Zeiger auf die Audiodaten Bits 16-18 von Kanal 1 Bits 0-15
\$0B2	AUD1LCL	
\$0C0	AUD2LCH	Zeiger auf die Audiodaten Bits 16-18 von Kanal 2 Bits 0-15
\$0C2	AUD2LCL	
\$0D0	AUD3LCH	Zeiger auf die Audiodaten Bits 16-18 von Kanal 3 Bits 0-15
\$0D2	AUD3LCL	

Die Initialisierung dieser Adreßzeiger kann wie üblich mit einem MOVE.L-Befehl erfolgen:

```
LEA $DFF000, A5 ;Basisadresse der Custom-Chips nach A5
MOVE.L #Start, AUD0LCH(A5) ;"Start" in AUD0LC schreiben
```

Als nächstes muß man dem DMA-Controller die Länge der digitalisierten Schwingung mitteilen, d.h. aus wie vielen Samples sie besteht. Die entsprechenden Register sind die AUDxLEN-Register:

Reg.	Name	Funktion
\$0A4	AUD0LEN	Anzahl der Audiodatenworte von Kanal 0
\$0B4	AUD1LEN	Anzahl der Audiodatenworte von Kanal 1
\$0C4	AUD2LEN	Anzahl der Audiodatenworte von Kanal 2
\$0D4	AUD3LEN	Anzahl der Audiodatenworte von Kanal 3

Die Länge wird aber nicht in Bytes, sondern in Worten angegeben. Daher muß die Anzahl der Bytes noch durch 2 geteilt werden, bevor man sie in das AUDxLEN-Register schreibt.

Die Initialisierung des AUDxLEN-Registers kann mit folgendem MOVE-Befehl erfolgen. Um das Abzählen der Worte zu sparen, wurden zwei Labels definiert: "Start" ist die Anfangsadresse der Datenliste, "End" die Endadresse+1 (siehe Beispieldatenliste oben). In A5 befindet sich die Basisadresse der Customchips (\$DFF000):

```
MOVE.W #(End-Start)/2, AUD0LEN(A5)
```

Jetzt kommt die Lautstärke des Tons. Beim Amiga kann man die Lautstärke für jeden Kanal getrennt einstellen. Dabei stehen 65 Stufen zur Verfügung. Von 0 (unhörbar) bis zu 64 (volle Lautstärke) reicht der Einstellbereich. Die entsprechenden Register heißen AUDxVOL:

Reg.	Name	Funktion
\$0A8	AUD0VOL	Lautstärke des Audiokanals 0
\$0B8	AUD1VOL	Lautstärke des Audiokanals 1
\$0C8	AUD2VOL	Lautstärke des Audiokanals 2
\$0D8	AUD3VOL	Lautstärke des Audiokanals 3

Setzen wir unseren Tonkanal auf halbe Lautstärke:

```
MOVE.W #32, AUD0VOL(A5)
```

Der letzte Parameter, der noch fehlt, ist die Sampling-Rate. Sie bestimmt, in welchen Zeitabständen jeweils ein Datenbyte, also ein Sample, an den Digital/Analog-Wandler ausgegeben wird. Die Sampling-Rate bestimmt damit die Frequenz des Tons. Wie ganz am Anfang erwähnt wurde, ist die Frequenz gleich der Anzahl der Schwingungen pro Sekunde. Eine Schwingung besteht aus einer frei wählbaren Anzahl Samples. In unserem Beispiel sind es 16. Wenn die Sampling-Rate die Anzahl der gelesenen Samples pro Sekunde darstellt, entspricht die Frequenz des Tons der Sampling-Rate dividiert durch die Anzahl der Samples pro Schwingung:

$$\text{Tonfrequenz} = \frac{\text{Sampling-Rate}}{\text{Samples pro Schwingung}}$$

Leider läßt sich die Sampling-Rate nicht direkt in Hertz angeben. Der DMA-Controller will statt dessen die Anzahl der Buszyklen wissen, die zwischen der Ausgabe zweier Samples liegen sollen. Ein Buszyklus dauert genau 279.365 Nanosekunden (Milliardstel Sekunden), in Sekunden ausgedrückt: $2.79365 \cdot 10^{-7}$ oder ausgeschrieben: 0.000000279365 Sekunden.

Um von der Sampling-Rate zu der Anzahl der Buszyklen zu kommen, bildet man den Kehrwert der Sampling-Rate. Dadurch erhält man die Dauer eines Samples. Dividiert man diesen Wert dann durch die Dauer eines Buszyklus in Sekunden erhält man die Anzahl der Buszyklen zwischen zwei Samples, die sogenannte Sampleperiod:

$$\text{Sampleperiod} = \frac{1}{\text{Sampling-Rate} * 2.79365 * 10^{-7}}$$

Nehmen wir einmal an, wir wollten unseren Beispielton mit einer Frequenz von 440 Hz spielen, also ein eingestrichenes "a". Die Sampling-Rate errechnet sich dann wie folgt:

$$\text{Sampling-Rate} = \text{Frequenz} * \text{Samples pro Schwingung}$$

$$\text{Sampling-Rate} = 440 \text{ Hz} * 16 = 7040 \text{ Hz}$$

Die notwendige Sampleperiod ist auch schnell da, wenn man die Werte in die zugehörige Gleichung einsetzt:

$$\text{Sampleperiod} = \frac{1}{7040 * 2.79365 * 10^{-7}} = 508.4583$$

Da man als Sampleperiod nur ganzzahlige Werte angeben kann, rundet man das Ergebnis auf 508 ab. Damit ist die Ausgangsfrequenz zwar nicht mehr exakt 440 Hz, aber die Abweichung bleibt minimal, nämlich 0.4 Hz.

Die Sampleperiod kann theoretisch Werte zwischen 0 und 65535 annehmen. Der tatsächliche Bereich ist aber nach oben hin begrenzt. Wie man der Abbildung 1.5.3.2 im Kapitel "Grundlagen" entnehmen kann, besitzt jeder Audiokanal einen DMA-Slot pro Rasterzeile, d.h. in jeder Rasterzeile können ein Datenwort bzw. zwei Samples aus dem Speicher gelesen werden. Daher ist der kleinstmögliche Wert für die Sampleperiod gleich 124. Die Samplefrequenz beträgt dann 28867 Hz. Verkürzt man die Sampleperiod über 124 hinaus, kann es vorkommen, daß ein Datenwort

zweimal ausgegeben wird, da das nächste nicht mehr rechtzeitig gelesen werden konnte.

Die Sampleperiod-Register heißen AUDxPER:

Reg.	Name	Funktion
\$0A6	AUD0PER	Sampleperiod für Audiokanal 0
\$0B6	AUD1PER	Sampleperiod für Audiokanal 1
\$0C6	AUD2PER	Sampleperiod für Audiokanal 2
\$0D6	AUD3PER	Sampleperiod für Audiokanal 3

MOVE.W #508,AUD0PER(A5) befördert die von uns errechnete Sampling-Rate in das AUD0PER-Register. Damit sind alle Register des Audiokanals Nummer 0 mit den richtigen Werten für unseren Ton versehen. Um ihn zum Erklängen zu bringen, muß man noch den DMA-Zugriff für den Audio-DMA-Kanal 0 einschalten. Vier Bits im DMACON-Register sind für die Audio-DMA-Kanäle zuständig:

DMACON-Bit Nr.	Name	Audio-DMA-Kanal Nr.
3	AUD3EN	3
2	AUD2EN	2
1	AUD1EN	1
0	AUD0EN	0

Soll der Audio-DMA für Kanal 0 eingeschaltet werden, muß das AUD0EN-Bit auf 1 gesetzt werden. Sicherheitshalber sollte man auch das DMAEN-Bit mitsetzen (siehe Kapitel "Grundlagen"):

```
MOVE.W #$8201, DMACON(A5) ;AUD0EN und DMAEN setzen
```

Jetzt beginnt der DMA-Controller, die Audiodaten aus dem Speicher zu holen und über den Digital/Analog-Wandler auszugeben. Der Ton ist im Lautsprecher zu hören. Will man ihn wieder abschalten, setzt man einfach AUD0EN = 0.

Immer wenn man AUDxEN auf 1 setzt, beginnt der DMA bei der Adresse in AUDxLC. Es gibt allerdings eine Ausnahme: War der DMA-Kanal an, also AUDxEN = 1, und man setzt das Bit

nur kurz auf 0 und dann wieder auf 1, ohne daß der DMA-Kanal in der Zwischenzeit ein neues Datenwort gelesen hätte, fährt der DMA-Controller an der alten Adresse fort.

Audio-Interrupts

Der Audio-DMA beginnt immer mit dem Datenbyte an der Adresse in AUDxLC. Sind soviel Datenworte aus dem Speicher geholt und ausgegeben worden, wie in AUDxLEN festgelegt, beginnt der DMA wieder bei der AUDxLC-Adresse. Im Gegensatz zu den Adreßregistern des Blitters oder der Bitplanes wird der Inhalt des AUDxLC-Registers während des gesamten Audio-DMA nicht verändert. Es existiert vielmehr für jeden Audiokanal noch ein zusätzliches Adreßregister. Bevor der DMA-Controller das erste Datenbyte aus dem Speicher holt, kopiert er den Wert des AUDxLC-Registers in dieses interne Adreßregister. Auch das AUDxLEN-Register überträgt er in einen internen Zähler. Ist dies geschehen, wird ein Interrupt ausgelöst. Wie man im Kapitel "Interrupts" nachlesen kann, existiert für jeden der vier Audiokanäle ein eigenes Interruptbit. Der Prozessor-Interrupt der Ebene 4 ist ausschließlich für diese Bits reserviert.

Während der DMA-Controller jetzt Datenwort um Datenwort aus dem Speicher holt, kann der Prozessor AUDxLC und AUDxLEN schon mit neuen Daten versorgen, da beide Register ja intern gespeichert sind. Erst wenn der Zähler, der am Anfang mit dem Wert von AUDxLEN initialisiert wurde, bei 0 ankommt, werden die Daten aus AUDxLC und AUDxLEN erneut gelesen. Der Prozessor hat dadurch genügend Zeit, die Werte der beiden Register, falls notwendig, zu ändern. Dadurch ist eine unterbrechungsfreie Tonausgabe möglich.

Nach jeder kompletten Schwingung wird also ein Interrupt ausgelöst. Bei hohen Tonfrequenzen treten dadurch sehr häufig Interrupts auf. Man sollte die Interrupt-Enable-Bits (INTEN) der Audio-Interrupts nur setzen, wenn man diese Interrupts auch wirklich benötigt. Der Prozessor kann sich sonst vor lauter Interrupt-Anforderungen kaum mehr retten.

Modulation von Lautstärke und Frequenz

Um bestimmte Klangeffekte zu erzeugen, besteht die Möglichkeit, Frequenz und/oder Lautstärke zu modulieren. Dabei arbeitet immer ein DMA-Kanal als Modulator, der die entsprechenden Parameter eines anderen Kanals verändert. Dies geschieht auf eine sehr einfache Weise: Der Modulationsoszillator holt wie üblich seine Daten aus dem Speicher. Aber statt sie an den Digital/Analog-Wandler auszugeben, schreibt er sie in das Lautstärke- oder Frequenzregister des Oszillators, den er modulieren soll (AUDxVOL oder AUDxLEN). Er kann auch beide Register gleichzeitig beeinflussen. In diesem Fall werden die Datenworte aus seiner Datenliste abwechselnd in das AUDxVOL- oder AUDxLEN-Register geschrieben. Die Datenworte haben das gleiche Format wie ihre Zielregister:

Volume: Bits 7-15 unbenutzt
 Bits 0-6 Lautstärkewert zwischen 0 und 64

Frequenz: Bits 0-15 Sampleperiod

Folgende Tabelle zeigt die Verwendung der Datenworte des Modulationsoszillators für alle drei möglichen Fälle:

Datenwort Nr.	Oszillator moduliert:		
	Frequenz	Lautstärke	Frequenz & Lautstärke
1	Period 1	Volume 1	Volume 1
2	Period 2	Volume 2	Period 1
3	Period 3	Volume 3	Volume 2
4	Period 4	Volume 4	Period 2

Um einen Audiokanal als Modulator zu aktivieren, muß man das/die entsprechenden Bits in dem Audio-Disk-Control-Register (ADKCON) setzen. Jeder Kanal kann nur seinen Nachfolger modulieren, d.h. Kanal 0 moduliert Kanal 1, 1 moduliert 2 und 2 moduliert 3. Auch Kanal 3 kann man als Modulator schalten, seine Datenworte werden aber nicht zur Modulation eines anderen Kanals verwendet und gehen verloren. Benutzt man einen Audio-Kanal als Modulator, wird sein Audioausgang abgeschaltet.

Das ADKCON-Register enthält, wie sein Name schon sagt, auch Steuerbits für den Diskcontroller, sie sind hier nicht aufgeführt und werden in dem entsprechenden Kapitel näher erklärt.

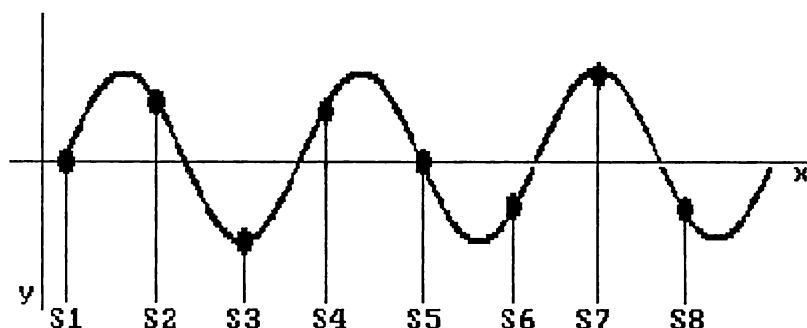
ADKCON-Register \$09E (schreiben) \$010 (lesen)

Bit Nr.	Name	Funktion
15	SET/CLR	Bits werden gesetzt (SET/CLR=1) oder gelöscht
14 bis 8		Steuerbits des Diskcontrollers
7	USE3PN	Audiokanal 3 moduliert nichts
6	USE2P3	Audiokanal 2 moduliert Period von Kanal 3
5	USE1P2	Audiokanal 1 moduliert Period von Kanal 2
4	USE0P1	Audiokanal 0 moduliert Period von Kanal 1
3	USE3VN	Audiokanal 3 moduliert nichts
2	USE2V3	Audiokanal 2 moduliert Volume v. Kanal 3
1	USE1V2	Audiokanal 1 moduliert Volume v. Kanal 2
0	USE0V1	Audiokanal 0 moduliert Volume v. Kanal 1

Um es noch einmal zu wiederholen: Verwendet man einen Kanal zur Modulation, werden lediglich seine Datenworte in die entsprechenden Register des zu modulierenden Kanals geschrieben. Sonst arbeiten beide weiterhin völlig unabhängig voneinander.

Probleme der digitalen Tonerzeugung des Amiga

Digitalisierung mehrerer Schwingungen zur Verbesserung der Qualität hoher Töne



S = Sample

Abb. 1.5.8.4

In unserem Beispiel hatten wir eine Schwingung mittels 16 Samples definiert. Die maximale Sampling-Rate ist 28867 Hz. Dies ergibt eine Höchstfrequenz von $28867 / 16 = 1460.4$ Hz. Das entspricht etwa einem dreigestrichenen Fis (1480 Hz).

Will man höher hinaus, muß man die Anzahl der Samples pro Schwingung verringern. Definieren wir unseren Sinus mit der halben Anzahl Samples, verdoppelt sich die Grenzfrequenz auf 3020.8 Hz. Acht Datenbytes sind aber etwas wenig für einen schönen Sinus. Für noch höhere Töne schrumpft die Anzahl der

Samples weiter. Bei 6041.6 Hz sind es nur noch vier. Da lassen sich dann kaum mehr unterschiedliche Wellenformen darstellen.

Allerdings fällt das beim Hören kaum auf. Denn unserem Ohr geht es in dieser Beziehung ganz genauso. Je höher die Frequenz, desto schwieriger wird es, verschiedene Klänge voneinander zu unterscheiden.

Trotzdem kann es die Klangqualität verbessern, wenn man bei hohen Frequenzen mehrere Schwingungen zur Definition der gewünschten Wellenform verwendet, wie es in Abbildung 1.5.8.4 zu sehen ist.

Der Tiefpaßfilter

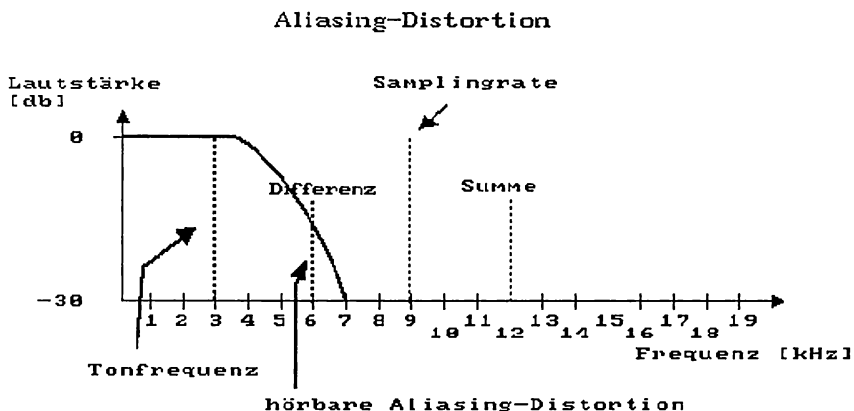
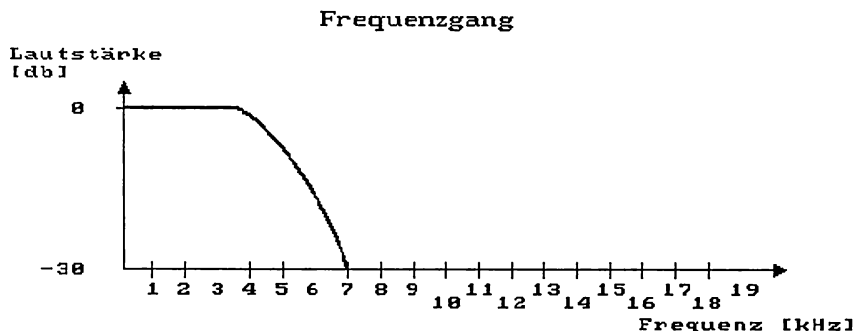


Abb. 1.5.8.5

Die Grenzfrequenz der Amiga-Tonausgabe wird aber noch von einem anderen Umstand eingeschränkt. Beim Analogisieren der digitalen Klangdaten entstehen nämlich zwei unerwünschte Störfrequenzen durch Wechselwirkungen zwischen der Sampling-Rate und der gewünschten Tonfrequenz. Die eine davon ist die Summe und die andere die Differenz aus Sampling-Rate und

Tonfrequenz. Dieses Phänomen trägt die Bezeichnung "Aliasing Distortion".

Bei einem Ton von beispielsweise 3 kHz und einer 12 kHz Sampling-Rate liegt die Differenz bei 9 und die Summe bei 15 kHz.

Um diese Störfrequenzen zu beseitigen, hat man einen sogenannten Tiefpaßfilter zwischen den Ausgängen der Digital/Analog-Wandler und den Audiobuchsen eingebaut. Seine Funktionsweise ist in Abbildung 1.5.8.5 dargestellt. Alle Frequenzen bis 4 kHz können ungestört passieren. Zwischen 4 und 7 kHz wird das Signal immer mehr abgeschwächt, bis oberhalb der 7 kHz keine Frequenzen mehr durchgelassen werden. Nehmen wir unser Zahlenbeispiel von oben: Der 3 kHz-Ton wird von dem Tiefpaß nicht abgeschwächt, aber sowohl Summen- als auch Differenzfrequenz liegen mit 9 bzw. 15 kHz über der Grenzfrequenz des Filters von 7 kHz und können ihn nicht mehr passieren. Damit sind sie auch nicht mehr im Lautsprecher zu hören. Versucht man aber, denselben 3 kHz Ton mit einer Sampling-Rate von 9 kHz auszugeben, ist die Differenzfrequenz $9 \text{ kHz} - 3 \text{ kHz} = 6 \text{ kHz}$ und wird damit zwar vom Filter abgeschwächt, aber noch durchgelassen.

Will man sichergehen, daß die Differenzfrequenz immer oberhalb der Grenzfrequenz des Filters bleibt, muß man sich an folgende Regel halten:

$$\text{Sampling-Rate} > \text{höchste Frequenzkomponente} + 7 \text{ kHz}$$

Es genügt nicht, wenn man dafür sorgt, daß die Differenz aus Samplingfrequenz und gewünschter Ausgangsfrequenz größer als 7 kHz ist. Wenn man eine Wellenform verwendet, die sehr viele Obertöne enthält, produziert jeder davon seine eigene Differenzfrequenz mit der Sampling-Rate. Aus diesem Grund muß man in den obigen Ausdruck immer die höchste Frequenzkomponente der verwendeten Wellenform einsetzen.

Der Tiefpaßfilter hält nicht nur die Störfrequenzen zurück, er beschränkt leider auch den Frequenzgang des Amiga. Zwar tre-

ten in einem Musikstück selten Töne mit einer Grundfrequenz zwischen 4 und 7 kHz auf, aber die Obertöne bestimmter Wellenformen liegen schon bei viel niedrigeren Grundfrequenzen innerhalb dieses Bereichs. Besonders deutlich wirkt sich das auf eine Rechteckschwingung aus. In Abbildung 1.5.8.2 kann man sehen, daß ein Rechteck aus der Kombination mehrerer Sinuswellen entsteht, die zueinander in einem festen Frequenzverhältnis stehen. In der Abbildung setzte sich das Rechteck lediglich aus den drei ersten Teiltönen zusammen. Ein tatsächliches Rechteck besteht dagegen aus unendlich vielen Teiltönen. Werden die hochfrequenten Obertöne durch den Filter abgeschnitten oder begrenzt, entsteht solch ein deformiertes Rechtecksignal wie in Abbildung 1.5.8.2. Im Extremfall, wenn die Grundfrequenz des Rechtecks der Grenzfrequenz des Filters nahekommt, kann es passieren, daß nur noch der erste Teilton übrigbleibt. Dann ist aus dem ursprünglichen Rechteck ein Sinus geworden.

Lautstärkeverlauf eines Tones

Der Klang eines Instruments wird außer von der Wellenform auch noch von seinem Lautstärkeverlauf bestimmt. In Punkto Wellenformen ist der Amiga ja fast zu allem fähig. Wie sieht es nun mit der Programmierung eines bestimmten Lautstärkeverlaufs aus?

Den Lautstärkeverlauf eines Tons kann man in drei Abschnitte aufteilen: den Anschlag, die Haltephase und das Abklingen.

Sobald der Ton gespielt wird, beginnt die Anschlagphase. Sie bestimmt, wie schnell sich die Lautstärke von null auf den Haltewert einpendelt. Während der Haltephase erklingt der Ton dann mit dieser Lautstärke. Wenn der Ton beendet wird, geht die Haltephase in die Abklingphase über, in der die Lautstärke dann vom Haltewert wieder auf null zurückgeht.

Diesen Verlauf kann man in Form einer Kurve, der sogenannten Hüllkurve, in ein Achsenkreuz eintragen.

Wie setzt man eine solche Hüllkurve auf den Amiga um?

Grundsätzlich gibt es drei Möglichkeiten:

1. Lautstärkemodulation

Man verwendet einen zweiten Tonkanal, um die Lautstärke des Tons zu modulieren. Z.B. Kanal 0 als Modulator von Kanal 1. Kanal 1 läßt man dabei ständig den gewünschten Ton ausgeben, seine Lautstärke setzt man aber auf null.

Die gewünschte Hüllkurve wird in zwei Abschnitte eingeteilt: Anschlagphase und Abklingphase. Ihr Verlauf wird digitalisiert (funktioniert genauso wie bei den Wellenformen) und in zwei Datenlisten im Speicher abgelegt. Soll der Ton gespielt werden, setzt man Kanal 0 auf die Adresse der Anschlagdaten und startet ihn. Da er die Lautstärke von Kanal 1 moduliert, folgt die Lautstärke des Tons genau der gewünschten Anschlagphase. Hat die Anschlagphase den Haltewert erreicht, ist die Datenliste von Kanal 0 abgearbeitet. Er erzeugt jetzt einen Interrupt und würde normalerweise die Datenliste noch einmal von vorne beginnen. Daher muß der Prozessor auf den Interrupt reagieren und mittels des AUD0EN-Bits im DMACON-Register Kanal 0 abschalten. Kanal 1 bleibt damit auf der gewünschten Haltelautstärke.

Soll der Ton wieder abgeschaltet werden, setzt man Kanal 0 auf die Abklingdaten und startet ihn erneut. Wieder wartet man auf den Interrupt, der anzeigt, daß die Abklingphase beendet ist, und schaltet Kanal 0 ab.

Die Register von Kanal 0 müssen bei diesem Vorgang wie folgt initialisiert werden:

- | | |
|---------|---|
| USE0V1 | Dieses Bit im ADKCON-Register setzt man auf 1, damit Kanal 0 die Lautstärke von Kanal 1 moduliert. |
| AUD0LC | Setzt man zuerst auf die Datenliste der Anschlag- und danach auf die der Abklingphase. |
| AUD0LEN | Enthält je nach der Adresse in AUD0LC einmal die Länge der Anschlag- und einmal die der Abklingdaten. |

- AUD0VOL Hat keine Funktion, da der Audioausgang von Kanal 0 abgeschaltet ist.
- AUD0PER Der Inhalt des AUD0PER-Registers bestimmt die Geschwindigkeit, mit der die Daten für die Lautstärke aus dem Speicher geholt werden. Man kann damit die Dauer der Anschlag- bzw. Abklingphase einstellen.

Mittels dieser Methode läßt sich die gewünschte Hüllkurve perfekt nachbilden. Leider ist sie aber mit einem großen Nachteil verbunden: Man benötigt zwei Audiokanäle für einen Ton. Will man weiterhin vier verschiedene Tonkanäle benutzen können, muß man die zweite Methode anwenden:

2. Steuerung der Lautstärke mittels des Prozessors

Die gewünschte Hüllkurve wird genau wie oben in den Speicher übertragen. Allerdings verändert man die Lautstärke diesmal mit dem Prozessor. Dieser holt in regelmäßigen Zeitabschnitten den aktuellen Lautstärkewert aus dem Speicher und schreibt ihn in das Lautstärkeregister des entsprechenden Tonkanals. Man muß das entsprechende Programm als Interrupt-Routine ablaufen lassen. Dies kann innerhalb des Vertical-Blanking-Interrupts geschehen, oder man verwendet einen der Timer-Interrupts von CIA-B.

Der Nachteil dieser Methode ist der Bedarf an Rechenzeit, da die Steuerung der Lautstärke diesmal nicht per DMA geschieht. Da sich dieser Bedarf allerdings in Grenzen hält, ist diese Methode für die meisten Anwendungsfälle am geeignetsten.

3. Einbau der Hüllkurve in die Schwingungsdaten

Diese Methode ist vor allem bei kurzen Klängen oder Geräuschimitationen vorteilhaft. Statt nur eine Schwingung der gewünschten Wellenform zu digitalisieren, schreibt man den gesamten Ablauf in den Speicher. Dieser kann entweder von einem Programm berechnet werden, oder man verwendet einen Audio-digitalisierer. Damit kann ein Geräusch mittels eines Mikrofons und eines Analog/Digital-Wandlers hardwaremäßig digitalisiert

werden. Solche Geräte werden von verschiedenen Firmen für den Amiga angeboten. Hat man die Daten dann im Amiga, kann man sie in jeder Tonhöhe bzw. Geschwindigkeit abspielen. Auf diese Weise kann man auch komplexe Effekte wie Gelächter oder Schreie täuschend echt mit dem Amiga nachahmen.

Auch diese Methode hat ihre Nachteile: Entweder erfordert sie schwierige Berechnungen oder zusätzliche Hardware, um den kompletten Klang in digitalisierter Form im Speicher abzulegen. Außerdem ist der Speicherplatzbedarf sehr groß. Dauert der Klang z.B. 1 Sekunde bei einer Sampling-Rate von 20 kHz, fressen die Klangdaten 20 KByte Speicher!

Tips, Tricks und Sonstiges

Klangqualität

Der Wertebereich der digitalen Daten reicht von -128 bis 127. Diesen Bereich sollte man möglichst ganz ausnutzen. Es ist am besten, wenn die Amplitude der digitalisierten Schwingung gleich 256 ist. Die Klangqualität kann sonst hörbar abnehmen, da die Größe des Quantisierungsfehlers mit der Verringerung des Wertebereichs zunimmt und mit ihm auch das Quantisierungsrauschen, das sehr schnell störende Ausmaße erreicht.

Man sollte es aus diesen Gründen vermeiden, die Amplitude der digitalisierten Schwingung zur Steuerung der Lautstärke zu verwenden. Dafür besitzt jeder Kanal sein AUDxVOL-Register. Verringert man damit die Lautstärke, bleibt das Verhältnis zwischen gewünschtem Klang und Störgeräuschen voll erhalten und damit auch die hohe Klangqualität des Amiga.

Störungsfreie Übergänge beim Wechsel der Wellenform

Um Tonstörungen wie Knackser oder Lautstärkesprünge beim Wechsel der Wellenform zu vermeiden, muß man folgende Regeln beachten:

Jede Schwingung sollte immer von Nulldurchgang zu Nulldurchgang digitalisiert werden, d.h. man beginnt bei einem Schnittpunkt der Schwingung mit der X-Achse, die Daten in den Speicher zu übertragen. Hält man sich an diese Regel, ist sichergestellt, daß alle Wellenformen im Speicher mit demselben Wert beginnen und enden, nämlich null. Dann können beim Aneinanderreihen unterschiedlicher Schwingungen keine plötzlichen Pegelsprünge auftreten, die als Knackser zu hören wären.

Als zweites muß man darauf achten, daß die Gesamtlautstärke der beiden Schwingungen annähernd gleich ist. Damit ist der sogenannte Effektivwert der Schwingung gemeint. Der Effektivwert einer Schwingung ist gleich der Amplitude eines Rechtecksignals, dessen Fläche unter der Kurve genauso groß wie diejenige der Schwingung ist.

Dieser Effektivwert bestimmt die Lautstärke einer Schwingung. Nur beim Rechteck ist er gleich der Amplitude. Wechselt man von einer Wellenform auf eine mit einem höheren Effektivwert, klingt diese lauter als ihre Vorgängerin.

Der Effektivwert einer Schwingung läßt sich aus ihren digitalisierten Daten einfach berechnen:

Man addiert die Beträge sämtlicher Bytes und dividiert sie anschließend durch die Anzahl der Datenbytes.

Will man bei allen Wellenformen den vollen acht Bit-Wertebereich der Digital/Analog-Wandler ausnützen, können sich ihre Effektivwerte nicht immer entsprechen. Man muß dann bei einem Wechsel der Wellenform die Lautstärke im AUDxVOL-Register entsprechend anpassen.

Spiele von Noten

Normalerweise wird ein Musikstück in Form von Noten aufgeschrieben. Will man ein solches auf dem Amiga spielen, muß man die Notenwerte in die entsprechende Sampleperiod umwandeln. Um sich umständliche Rechnungen zu ersparen, verwendet

man am besten eine Tabelle, die die Sampleperiod-Werte für sämtliche Halbtöne einer Oktave enthält:

Tabelle der Sampleperiod-Werte von Musiknoten:

Note Frequenz [Hz] Sampleperiod bei AUDxLEN = 16

c	261.7	427	(262.0)
c [#]	277.2	404	(276.9)
d	293.7	381	(293.6)
d [#]	311.2	359	(311.6)
e	329.7	339	(330.0)
f	349.3	320	(349.6)
f [#]	370.0	302	(370.4)
g	392.0	285	(392.5)
g [#]	415.3	269	(415.8)
a	440.0	254	(440.4)
a [#]	466.2	240	(466.0)
h	493.9	226	(495.0)
c	523.3	214	(522.7)

(Die Werte in Klammern stellen die tatsächliche Frequenz bei der entsprechenden Samplingperiod dar.)

Noch eine Anmerkung zur Berechnung obiger Werte. Die Frequenz eines Halbtons ist immer um den Faktor "zwölfte Wurzel aus 2" höher als die des Vorgängers. $440 (a) * 2^{(1/12)} = 466.2 (a^{\#})$, $466.2 (a^{\#}) * 2^{(1/12)} = 493.9 (h)$ usw.

Eine Oktave entspricht immer einer Frequenzverdoppelung.

Will man jetzt eine Note aus einer Oktave spielen, die nicht in der Tabelle enthalten ist, gibt es zwei Möglichkeiten:

1. Man ändert die Samplingperiod. Für jede Oktave nach oben muß man den Wert halbieren. Eine Oktave tiefer entspricht der doppelten Samplingperiod. Dies ist zwar einfach, man stößt aber schnell an gewisse Grenzen. Bei einem Datenfeld von 32 Byte (AUDxLEN = 16), wie in unserer Tabelle, ist die kleinstmögliche Samplingperiod

(124) schon mit dem zweigestrichenen "a" erreicht. Man muß also die Datenliste verkleinern.

In diesem Fall bekommt man aber Probleme im Bereich der tiefen Töne, da die Störfrequenzen der Aliasing Distortion (siehe entsprechenden Abschnitt) hörbar werden.

Eine bessere Lösung stellt daher Verfahren Nr. 2 dar:

2. Man erstellt für jede Oktave eine eigene Datenliste. Der Samplingperiod-Wert bleibt dabei für jede Oktave konstant. Er dient nur zur Wahl des Halbtons. Soll ein Ton eine Oktave über demjenigen in der Tabelle liegen, verwendet man eine Datenliste, die nur noch halb so lang ist. Entsprechend eine doppelt so lange für die nächsttiefere Oktave.

Der normale Tonumfang beträgt etwa 8 Oktaven, d.h. man benötigt acht Datenlisten pro Wellenform.

Als Ausgleich für den höheren Aufwand erhält man mit diesem Verfahren unabhängig von der Tonhöhe immer den optimalen Klang.

Erzeugen hoher Frequenzen

Die minimale Samplingperiod beträgt normalerweise 124. Der Grund dafür ist, daß der Audio-DMA innerhalb einer noch kürzeren Samplingperiod nicht mehr in der Lage wäre, die Datenworte rechtzeitig zu lesen. Das alte Datenwort wird dann mehrfach ausgegeben. Diesen Effekt kann man sinnvoll nutzen. Da das gelesene Datenwort zwei Samples enthält, kann mit ihm ein Rechtecksignal von hoher Frequenz erzeugt werden. Bei einer Samplingperiod von 1 ergibt sich eine Samplingfrequenz von 3.58 MHz und eine Ausgangsfrequenz von 1.74 MHz! Um dieses hochfrequente Ausgangssignal auch nutzen zu können, muß man es vor dem Tiefpaßfilter abgreifen. Dazu kann man den AUDIN-Eingang (Pin 16) der seriellen Buchse (RS232) ver-

wenden. Er ist direkt mit dem rechten Audioausgang von Paula verbunden (siehe Kapitel Schnittstellen).

Um solch hohe Frequenzen erzeugen zu können, muß AUDxVOL auf volle Lautstärke gesetzt werden (AUDxVOL = 64).

Spiele mehrstimmiger Musik

Da der Amiga vier unabhängige Audiokanäle besitzt, ist es problemlos möglich, vier verschiedene Töne gleichzeitig zu erzeugen. Damit kann man alle vierstimmigen Musikstücke direkt spielen.

Aber es geht noch mehr. Denn vier Audiokanäle heißt noch lange nicht, daß vier Stimmen das Maximum sind. Es wurde ja schon erwähnt, daß jede Wellenform in Wirklichkeit eine Kombination aus Sinussignalen darstellt. Genauso wie diese Obertöne zusammen die Wellenform ergeben, kann man durch Kombination mehrerer Wellenformen einen mehrstimmigen Klang erzeugen. Die Ausgangssignale der Audiokanäle 0 und 3 werden ja auch innerhalb von Paula zu einem Stereokanal zusammen gemischt. Dabei werden die Wellenformen beider Kanäle zu einem einzigen, jetzt zweistimmigen Signal kombiniert.

Was mit den Analogsignalen elektronisch möglich ist, läßt sich aber auch mit den digitalen Daten rechnerisch erledigen! Man addiert die digitalen Daten zweier völlig verschiedener Wellenformen und gibt die neuen Daten wie üblich über den Audiokanal aus. Schon hat man zwei Stimmen pro Audiokanal. Auf diese Weise lassen sich theoretisch beliebig viele Stimmen über einen Tonkanal abspielen.

Praktisch ist diese Anzahl allerdings von der Rechengeschwindigkeit begrenzt, aber 16 Stimmen sind durchaus machbar!

Die Errechnung des Summensignals aus den Teilsignalen ist sehr einfach. Zu jedem Zeitpunkt werden die aktuellen Werte aller Töne aufaddiert und das Ergebnis durch ihre Anzahl dividiert. Auf ähnliche Weise entsteht auch ein Rechtecksignal, wenn man

Sinussignale im richtigen Frequenzverhältnis addiert (Abbildung 1.5.8.2).

Audioausgabe ohne DMA

Wie bei allen DMA-Kanälen, existieren auch beim Audio-DMA Datenregister, in die der DMA-Kanal die Daten überträgt und die ebenso vom Prozessor beschrieben werden können:

Die Audiodatenregister

Reg.	Name	Funktion
\$0AA	AUD0DAT	Diese vier Register enthalten immer das aktuelle Audiodatenwort, bestehend aus zwei Samples. Das Sample in dem oberen Byte (Bits 8-15) wird immer zuerst ausgegeben.
\$0BA	AUD1DAT	
\$0CA	AUD2DAT	
\$0DA	AUD3DAT	

Um die Audiodatenregister mit dem Prozessor beschreiben zu können, schaltet man den DMA mit AUDxEN = 0 ab. Dadurch ändert sich auch die Erzeugung der Audio-Interrupts. Sie treten jetzt immer nach der Ausgabe der beiden Samples im AUDxDAT-Register auf, statt wie früher zum Beginn jeder Audiodatenliste.

Lädt man nicht rechtzeitig ein neues Datenwort in AUDxDAT, werden die beiden letzten Samples entgegen dem DMA-Betrieb nicht wiederholt, sondern der Ausgang bleibt auf dem Wert des letzten Datenbytes (der unteren Hälfte des Worts in AUDxDAT) stehen.

Die direkte Programmierung der Audiodatenregister kostet sehr viel Rechenzeit. Außer in Spezialfällen sollte man lieber den Audio-DMA verwenden.

Ein paar Fakten

AUDxVOL-Werte in Dezibel (0 dB = volle Lautstärke):

AUDxVOL	dB	AUDxVOL	dB	AUDxVOL	dB	AUDxVOL	dB
64	0.0	48	-2.5	32	-6.0	16	-12.0
63	-0.1	47	-2.7	31	-6.3	15	-12.6
62	-0.3	46	-2.9	30	-6.6	14	-13.2
61	-0.4	45	-3.1	29	-6.9	13	-13.8
60	-0.6	44	-3.3	28	-7.2	12	-14.5
59	-0.7	43	-3.5	27	-7.5	11	-15.3
58	-0.9	42	-3.7	26	-7.8	10	-16.1
57	-1.0	41	-3.9	25	-8.2	9	-17.0
56	-1.2	40	-4.1	24	-8.5	8	-18.1
55	-1.3	39	-4.3	23	-8.9	7	-19.2
54	-1.5	38	-4.5	22	-9.3	6	-20.6
53	-1.6	37	-4.8	21	-9.7	5	-22.1
52	-1.8	36	-5.0	20	-10.1	4	-24.1
51	-2.0	35	-5.2	19	-10.5	3	-26.6
50	-2.1	34	-5.5	18	-11.0	2	-30.1
49	-2.3	33	-5.8	17	-11.5	1	-36.1

(AUDxVOL = 0 entspricht einem dB-Wert von minus unendlich)

Wenn AUDxVOL = 64 ist, dann entspricht ein digitaler Wert von 127 einer Ausgangsspannung von etwa 400 Millivolt, -128 entsprechen -400 Millivolt. Eine Änderung um 1 LSB bewirkt eine Schwankung der Ausgangsspannung von ca. 3 Millivolt.

Beispielprogramme*Programm 1: Erzeugen eines einfachen Sinustons*

Dieses Programm erzeugt einen Sinuston mit einer Frequenz von 440 Hz. Dabei wird die gleiche Sample-Tabelle wie im Text benutzt. Der größte Teil des Programms dient wieder der Anforderung von Chip-RAM für die Audiodatenliste.

Der Ton wird über Kanal 0 ausgegeben, bis die Maustaste gedrückt wird. Danach gibt das Programm den belegten Speicher zurück.

;*** Erzeugen eines einfachen Sinustons ***

;Customchip-Register

INTENA = \$9A ;Interruptenable-Register (schreiben)

DMACON = \$96 ;DMA-Controllregister (schreiben)

;Audio-Register

AUD0LC = \$A0 ;Adresse der Audiodatenliste

AUD0LEN = \$A4 ;Länge der Audiodatenliste

AUD0PER = \$A6 ;Samplingperiod

AUD0VOL = \$A8 ;Lautstärke

ADKCON = \$9E ;Steuerregister für Modulation

;CIA-A Portregister A (Maustaste)

CIAAPRA = \$bfe001

;Exec Library Base Offsets

AllocMem = -30-168 ;ByteSize,Requirements/d0,d1

FreeMem = -30-180 ;MemoryBlock,ByteSize/a1,d0

;sonstige Labels

Execbase = 4

chip = 2 ;Chip-RAM anfordern

;*** Vorprogramm ***

start:

;Speicher für Audiodatenliste anfordern

move.l Execbase,a6

moveq #ALsize,d0 ;Größe der Audiodatenliste

moveq #chip,d1

jsr AllocMem(a6) ;Speicher anfordern

beq Ende ;Fehler -> Programm beenden

```
;Audiodatenliste ins Chip-RAM kopieren

move.l d0,a0 ;Adresse im Chip-RAM
move.l #ALstart,a1 ;Adresse im Programm
moveq #ALsize-1,d1 ;Schleifenzähler

Loop: move.b (a1)+,(a0)+ ;Datenliste ins Chip-RAM
dbf d1,Loop

;*** Hauptprogramm

;Audioregister initialisieren

lea $DFF000,a5
move.w #$000f,dmacon(a5) ;Audio-DMA aus
move.l d0,aud0lc(a5) ;Adresse der Datenliste setzen
move.w #ALsize/2,aud0len(a5) ;Länge in Worten
move.w #32,aud0vol(a5) ;halbe Lautstärke
move.w #508,aud0per(a5) ;Frequenz: 440 Hz

move.w #$00ff,adkcon(a5) ;Modulation ausschalten

;Audio-DMA einschalten

move.w #$8201,dmacon(a5) ;Kanal 0 an

;Auf Maustaste warten

wait: btst #6,ciaapra
bne wait

;Audio-DMA ausschalten

move.w #$0001,dmacon(a5) ;Kanal 0 aus

*** Nachprogramm ***

move.l d0,a1 ;Adresse der Datenliste
moveq #ALsize,d0 ;Länge
jsr FreeMem(a6) ;belegten Speicher freigeben

Ende: clr.l d0
rts
```



```
;Audiodatenliste
```

```
ALstart:
```

```
dc.b 0,49
```

```
dc.b 90,117
```

```
dc.b 127,117
```

```
dc.b 90,49
```

```
dc.b 0,-49
```

```
dc.b -90,-117
```

```
dc.b -127,-117
```

```
dc.b -90,-49
```

```
ALend:
```

```
ALsize = ALend - ALstart ;Länge der Audiodatenliste
```

```
;Programmende
```

Programm 2: Sinuston mit Vibrato

Dieses Programm stellt eine Erweiterung des vorangegangenen dar. Es wird der gleiche Sinuston ausgegeben, diesmal allerdings über Kanal 1. Kanal 0 moduliert die Frequenz von Kanal 1 und erzeugt so ein Vibrato. Die Daten für das Vibrato stellen eine digitalisierte Sinusschwingung dar, deren Nullpunkt den Wert der Samplingperiod eines eingestrichenen "a", also 508, hat.

```
;*** Erzeugen eines Vibratos ***
```

```
;Customchip-Register
```

```
INTENA = $9A ;Interruptenable-Register (schreiben)
```

```
DMACON = $96 ;DMA-Controllregister (schreiben)
```

```
;Audio-Register
```

```
AUD0LC = $A0 ;Adresse der Audiodatenliste
```

```
AUD0LEN = $A4 ;Länge der Audiodatenliste
```

```
AUD0PER = $A6 ;Samplingperiod
```

```
AUD0VOL = $A8 ;Lautstärke
```

```
AUD1LC = $B0
```

```
AUD1LEN = $B4
```

```
AUD1PER = $B6
```

```
AUD1VOL = $B8
```

```

ADKCON = $9E ;Steuerregister für Modulation

;CIA-A Portregister A (Maustaste)

CIAAPRA = $bfe001

;Exec Library Base Offsets

AllocMem = -30-168 ;ByteSize,Requirements/d0,d1
FreeMem = -30-180 ;MemoryBlock,ByteSize/a1,d0

;sonstige Labels

Execbase = 4
chip = 2 ;Chip-RAM anfordern

;*** Vorprogramm ***
start:

;Speicher für Datenlisten anfordern

move.l Execbase,a6
move.l #Size,d0 ;Länge beider Listen
moveq #chip,d1
jsr AllocMem(a6) ;Speicher anfordern
beq Ende

;Audiodatenliste ins Chip-RAM kopieren

move.l d0,a0 ;Adresse im Chip-RAM
move.l #ALstart,a1 ;Adresse im Programm
move.w #Size-1,d1 ;Schleifenzähler

Loop: move.b (a1)+,(a0)+ ;Listen ins Chip-RAM
dbf d1,Loop

;*** Hauptprogramm

;Audioregister initialisieren

move.l d0,d1 ;Adresse der Audiodatenliste
add.l #ALsize,d1 ;Adresse der Vibratotabelle
lea $DFF000,a5
move.w #$000f,dmacon(a5) ;Audio-DMA aus
move.l d1,aud0lc(a5) ;auf Vibratotabelle setzen
move.w #Vibsize/2,aud0len(a5) ;Länge der Vibratotabelle

```

```
move.w #8961,aud0per(a5) ;Vibratofrequenz

move.l d0,aud1lc(a5) ;Kanal 1 auf Audiodatenliste
move.w #ALsize/2,aud1len(a5) ;Länge der Audiodatenliste
move.w #32,aud1vol(a5) ;halbe Lautstärke

move.w #$00FF,adkcon(a5) ;sonstige Modulation aus
move.w #$8010,adkcon(a5) ;Kanal 0 moduliert Period
;von Kanal 1
;Audio-DMA einschalten

move.w #$8203,dmacon(a5) ;Kanäle 0 und 1 an

;Auf Maustaste warten

wait: btst #6,ciaapra
bne wait

;Audio-DMA ausschalten

move.w #$0003,dmacon(a5) ;Kanäle 0 und 1 aus

*** Nachprogramm ***

move.l d0,a1 ;Adresse der Listen
move.l #Size,d0 ;Länge
jsr FreeMem(a6) ;Speicher freigeben

Ende: clr.l d0
rts

;Audiodatenliste

ALstart:
dc.b 0,49
dc.b 90,117
dc.b 127,117
dc.b 90,49
dc.b 0,-49
dc.b -90,-117
dc.b -127,-117
dc.b -90,-49
ALend:
ALsize = ALend - ALstart ;Länge der Audiodatenliste
```

```
;Vibratotabelle
```

```
Vibstart:
```

```
dc.w 508,513,518,522,524,525,524,522,518,513
```

```
dc.w 508,503,498,494,492,491,492,494,498,503
```

```
Vibend:
```

```
Vibsize = Vibend - Vibstart ;Länge der Vibratotabelle
```

```
Size = ASize + Vibsize ;Gesamtlänge beider Listen
```

```
;Programmende
```

1.5.9 Maus, Joystick und Paddles

Maus, Joystick und Paddles, all dies kann man an den Amiga anschließen. Gehen wir sie der Reihe nach durch, zusammen mit den zugehörigen Registern. Die Belegung der Gameports, an die all diese Eingabegeräte angeschlossen werden, kann man im Schnittstellenkapitel nachlesen. Beginnen wir mit der Maus:

Die Maus

Die Maus ist das meistgebrauchte Eingabegerät. Erst durch sie wird der Amiga so richtig schön. Aber wie funktioniert es, daß der Pfeil immer den Bewegungen der Maus folgt?

Dreht man die Maus um, erkennt man eine gummibeschichtete Stahlkugel, die sich beim Schieben der Maus entsprechend dreht. Diese Drehungen der Rollkugel werden auf zwei Achsen übertragen, die im rechten Winkel zueinander stehen. Damit sind sie so angeordnet, daß sich die eine bei Bewegungen entlang der X-Achse und die andere bei solchen in Richtung der Y-Achse dreht. Verschiebt man die Maus in diagonalen Richtung, drehen sich beide Achsen entsprechend der X- und Y-Komponenten der Mausbewegung.

Leider helfen drehende Achsen dem Amiga wenig, wenn er die Mausposition bestimmen will. Eine Umwandlung der mechanischen Bewegung in elektrische Signale ist notwendig.

Dazu ist am Ende jeder Achse eine Lochscheibe angebracht. Beim Drehen unterbricht sie immer wieder den Strahl einer Lichtschranke. Das dabei entstehende Signal wird verstärkt und über das Mauskabel zum Computer geleitet.

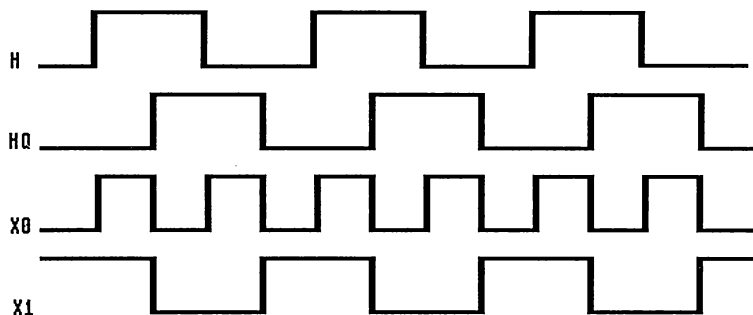
Jetzt ist der Amiga in der Lage festzustellen, ob und mit welcher Geschwindigkeit die Maus bewegt wird. Aber er weiß noch nicht, in welche Richtung, d.h. ob nach rechts oder links, hinten oder vorne.

Ein kleiner Trick löst dieses Problem. An jeder Lochscheibe werden zwei Lichtschranken angebracht, die gegeneinander um ein halbes Loch versetzt sind. Dreht sich die Scheibe in eine bestimmte Richtung, wird die eine Lichtschranke immer vor der anderen unterbrochen. Kehrt man die Bewegungsrichtung um, ändert sich auch die Lage der beiden Signale der Lichtschranken entsprechend. Damit kann der Amiga jetzt auch die Bewegungsrichtung der Maus bestimmen.

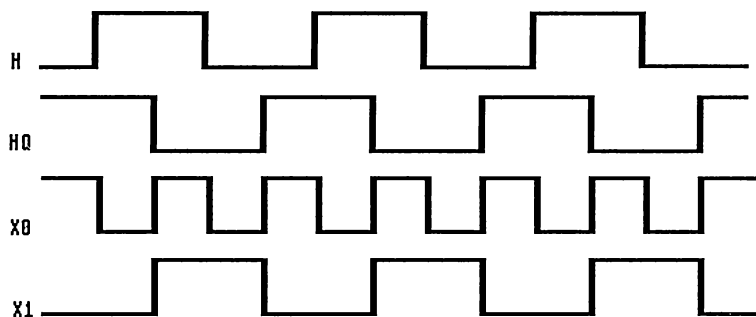
Die Maus liefert also vier Signale, zwei pro Achse. Sie tragen die Namen "Vertical Pulse", "Vertical Quadrature Pulse", "Horizontal Pulse" und "Horizontal Quadrature Pulse".

Die Maussignale

1. Bewegung nach rechts (Aufwärtszählen)



2. Bewegung nach links (Abwärtszählen)



Wahrheitstabellen für X0 und X1

H	HQ	X0
0	0	0
0	1	1
1	0	1
1	1	0

HQ	X1
0	1
1	0

Abb. 1.5.9

Die Abbildung 1.5.9 zeigt die Phasenlage der horizontalen "Pulse" (H) und "Quadrature Pulse" Signale (Q), ist aber für die vertikalen ebenso gültig. Man erkennt deutlich, wie sich je nach Bewegungsrichtung H und HQ gegeneinander verschieben. Aus ihnen gewinnt der Amiga durch logische Verknüpfungen zwei neue Signale, X0 und X1. X1 ist das invertierte Abbild von HQ, und X0 entsteht durch ein exclusives Oder von H und HQ, d.h. X0 wird immer dann 1, wenn H und HQ auf unterschiedlichen Pegeln liegen (siehe Wahrheitstabelle Abbildung 1.5.9). Mit diesen beiden Signalen steuert der Amiga einen 6-Bit-Zähler an, der je nach Richtung durch X1 hinauf oder herunter gezählt wird. Zusammen mit X0 und X1 entsteht so ein gemeinsamer 8-Bit-Wert, der die aktuelle Mausposition repräsentiert.

Bewegt man die Maus nach rechts bzw. unten, wird er hinaufgezählt. Bewegt man die Maus nach links bzw. oben, wird er heruntergezählt.

Es existieren vier derartige Zähler innerhalb von Denise. Zwei pro Gameport, denn es kann an jeden davon eine Maus angeschlossen werden. Sie heißen JOYDAT0 und JOYDAT1:

*JOY0DAT \$00A**(Maus an Gameport 0)**JOY1DAT \$00C**(Maus an Gameport 1)*

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Funktion:	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0

(Beide Register sind Nur-Lese-Register)**Y0-7** Zähler für die vertikalen Mausbewegungen (Y-Richtung)**H0-7** Zähler für die horizontalen Mausbewegungen (X-Richtung)

Die Maus erzeugt zweihundert Zählimpulse pro Inch oder umgerechnet ca. 79 pro Zentimeter, d.h. die Grenze der Mauszähler ist schnell erreicht. 8 Bit ergeben einen Zählbereich von 0 bis 256. Man muß die Maus nur um knapp vier Zentimeter verschieben, um einen Zählerüberlauf hervorzurufen. Dieser kann sowohl beim Hinaufzählen (der Zähler springt von 255 auf 0) als auch beim Herunterzählen (Sprung von 0 auf 255) stattfinden. Aus diesem Grund muß man die Zählregister in bestimmten Abständen abfragen und dabei prüfen, ob ein Über- bzw. Unterlauf stattgefunden hat.

Das Betriebssystem erledigt dies gewöhnlich innerhalb des Vertical-Blanking-Interrupts.

Man geht dabei davon aus, daß die Maus in der Zeit zwischen zwei Abfragen nicht weiter als 127 Zählschritte bewegt wurde. Dann vergleicht man den neuen Zählerstand mit dem zuletzt gelesenen. Ist die Differenz größer 127, hatte der Zähler einen Überlauf, und die Maus wurde nach rechts bzw. unten bewegt. Ist er kleiner -127, lag ein Unterlauf vor, entsprechend einer Mausbewegung nach links bzw. oben.

Alter Zählerstand	Neuer Zählerstand	tatsächliche Differenz	Mausbewegung	Unter/ Überlauf
100	200	-100	+100	nein
200	100	+100	-100	nein
50	200	-150	-105	Unterlauf
200	50	+150	+105	Überlauf

Differenz = Alter Zählerstand - Neuer Zählerstand

Trat ein Unterlauf auf, errechnet sich die tatsächliche Mausbewegung wie folgt:

$$-255 - \text{Differenz, oder in Zahlen: } -255 - (50-200) = -105$$

Bei einem Überlauf gilt:

$$255 - \text{Differenz, oder in Zahlen: } 255 - (200-50) = +105$$

Eine positive Mausbewegung entspricht einer Verschiebung nach rechts bzw. oben, ein negativer Wert nach links bzw. unten.

Die Mauszähler lassen sich auch per Software setzen. Mittels des JOYTEST-Registers kann man einen beliebigen Wert in die Zähler schreiben. JOYTEST wirkt immer auf beide Gameports gleichzeitig, d.h. sowohl die horizontalen als auch die vertikalen Zähler beider Mäuse werden mit demselben Wert initialisiert (JOY0DAT = JOY1DAT).

JOYTEST \$036 (nur schreiben)

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Funktion:	Y7	Y6	Y5	Y4	Y3	Y2	xx	xx	X7	X6	X5	X4	X3	X2	xx	xx

Wie man sieht, lassen sich nur die 6 Bits der Zähler beeinflussen. Dies ist logisch, wenn man sich daran erinnert, daß die unteren beiden Bits direkt aus den Maussignalen gewonnen werden, sich also nicht in einem internen Speicher befinden, den man verändern könnte.

Die Joysticks

Wenn man die Belegung der Gameports betrachtet, sieht man, daß die vier Richtungen der Joysticks die gleichen Leitungen wie die Mäuse belegen. Es ist daher naheliegend, daß sie auch mittels desselben Registers abgefragt werden. Tatsächlich werden die Joystickleitungen genauso wie die Maussignale bearbeitet, d.h. je zwei Leitungen werden zu den X0- und X1-Bits bzw. zu den Y0- und Y1-Bits kombiniert. Bei der Joystick-Abfrage muß man aus ihnen die Joystick-Stellung ermitteln:

Joystick nach rechts	X1 = 1	(Bit 1 JOYxDAT)
Joystick nach links	Y1 = 1	(Bit 9 JOYxDAT)
Joystick nach hinten	X0 EOR X1 = 1	(Bits 0 u. 1 JOYxDAT)
Joystick nach vorne	Y0 EOR Y1 = 1	(Bits 8 u. 9 JOYxDAT)

Um festzustellen, ob der Joystick nach vorne oder hinten gedrückt ist, muß man eine exklusive Oder-Verknüpfung aus X0 und X1 bzw. Y0 und Y1 bilden. Ist deren Ergebnis gleich eins, befindet sich der Joystick in der betreffenden Position. Folgendes Assemblerprogramm erledigt die Joystick-Abfrage für Gameport 1:

```
TestJoystick:
MOVE.W $DFF00C, D0 ;JOY1DAT nach D0 übertragen
BTST #1, D0 ;Bit Nr.1 testen
BNE RECHTS ;gesetzt? Wenn ja, Joystick rechts
BTST #9, D0 ;Bit Nr.9 testen
BNE LINKS ;gesetzt? Wenn ja, Joystick links
MOVE.W D0,D1 ;Kopie von D0 nach D1
LSR.W #1,D1 ;Y1 und X1 auf Position von Y0 und X0
EOR.W D0,D1 ;exklusiv Oder: Y1 EOR Y0 und X1 EOR X0
BTST #0, D1 ;Ergebnis von X1 EOR X0 testen
BNE HINTEN ;gleich 1? Wenn ja, Joystick hinten
BTST #8, D1 ;Ergebnis von Y1 EOR Y0 testen
BNE VORNE ;gleich 1? Wenn ja, Joystick vorne
BRA MITTE ;Joystick ist in Mittelstellung
```

Die Exklusiv-Oder-Verknüpfung wird in diesem Programm folgendermaßen ausgeführt:

Eine Kopie des JOY1DAT-Registers wird nach D1 gebracht und dort um ein Bit nach rechts geschoben. Damit haben X1 in D1 und X0 in D0 die gleiche Bitposition, ebenso Y1 und Y0. Ein EOR zwischen D0 und D1 verknüpft sowohl Y0 mit Y1 als auch X0 mit X1. Danach muß das Ergebnis in D1 nur noch mit den entsprechenden BTST-Befehlen überprüft werden.

Dieses Programm unterstützt keine diagonalen Joystickstellungen.

Die Paddles

Der Amiga besitzt pro Gameport zwei Analogeingänge, an die man veränderliche Widerstände, sogenannte Potentiometer, anschließen kann. Diese haben in jeder Stellung einen bestimmten Widerstand, den die Hardware in Paula ermitteln kann. Ein Paddle enthält ein solches Potentiometer, das sich mit einem Drehknopf verstellen läßt. Auch Analog-Joysticks arbeiten auf diese Weise. Je ein Potentiometer für die X- und Y-Richtung gibt die genaue Joystick-Stellung wieder.

Zwei Register enthalten die vier acht Bit-Werte der Analog-eingänge, POT0DAT für Gameport 0 und POT1DAT für Gameport 1:

POT0DAT \$012 POT1DAT \$014

Bit Nr.:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Funktion:	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0	X7	X6	X5	X4	X3	X2	X1	X0

(Beide Register sind Nur-Lese-Register)

Wie funktioniert die Widerstandsmessung?

Da ein Computer nur digitale Signale verarbeiten kann, benötigt er eine spezielle Schaltung, wenn er Analogsignale verarbeiten soll. Beim Amiga funktioniert die Bestimmung der externen Widerstandswerte folgendermaßen:

Die Potentiometer sollten einen maximalen Widerstandswert von 470 Kilo-Ohm ($\pm 10\%$) haben. Es ist auf der einen Seite mit dem +5-Volt-Ausgang und auf der anderen mit einem der vier Paddle-Eingänge der Gameports verbunden. Diese führen intern zu den entsprechenden Eingängen von Paula und zu vier Kondensatoren, die zwischen Eingang und Masse geschaltet sind.

Mittels eines speziellen Startbits wird die Messung begonnen. Paula legt dann alle Paddle-Eingänge kurz auf Masse und entlädt dadurch die Kondensatoren. Außerdem löscht es die Zähler in den POTxDAT-Registern. Danach zählt es mit jeder Bildschirmzeile die Zähler um eins hinauf, während die Kondensatoren über die Widerstände langsam wieder aufgeladen werden. Überschreitet die Kondensatorspannung einen bestimmten Wert, wird der zugehörige Zähler gestoppt. Auf diese Weise entspricht der Zählerstand genau der Größe des eingestellten Widerstands. Kleine Widerstände ergeben niedrige Zählerstände, große entsprechend hohe.

Das Startbit befindet sich in dem POTGO-Register:

POTGO \$034 (nur schreiben) POTGOR \$016 (nur lesen)

Bit Nr.	Name	Funktion
15	OUTRY	Gameport 1 POTY auf Ausgang umschalten
14	DATRY	Gameport 1 POTY Datenbit
13	OUTRX	Gameport 1 POTX auf Ausgang umschalten
12	DATRX	Gameport 1 POTX Datenbit
11	OUTLY	Gameport 0 POTY auf Ausgang umschalten
10	DATLY	Gameport 0 POTY Datenbit
9	OUTLX	Gameport 0 POTX auf Ausgang umschalten
8	DATLX	Gameport 0 POTX Datenbit
7 bis 1		unbelegt
0	START	Kondensatoren entladen und Messung starten

(Ein Schreibzugriff auf POTGO löscht beide POTxDAT-Register.)

Normalerweise setzt man das START-Bit innerhalb der vertikalen Austastlücke auf 1. Während das Bild dargestellt wird, laden sich die Kondensatoren auf, erreichen den Sollwert und stoppen die Zähler. In der nächsten vertikalen Austastlücke kann man dann die gültigen Potentiometerstellungen in den POTxDAT-Registern lesen.

Die vier Analogeingänge lassen sich wahlweise auch als normale digitale Ein-/Ausgangsleitungen programmieren. Die entsprechenden Steuer- und Datenbits befinden sich zusammen mit dem START-Bit im POTGO-Register. Mittels der OUTxx-Bits (OUTxx = 1) kann man jede Leitung getrennt auf Ausgang umschalten. Dadurch wird sie von der Steuerschaltung der Kondensatoren getrennt, und der Wert im DATxx-Bit von POTGO wird über sie ausgegeben.

Beim Lesen des DATxx-Bits in POTGOR erhält man immer den aktuellen Zustand der betreffenden Leitung.

Verwendet man die Analogports als Ausgang, muß man folgendes beachten:

Da die vier Analogports intern mit den Kondensatoren für die Widerstandsmessung verbunden sind (47 nF), kann es bis zu 300 Mikrosekunden dauern, bis die Leitung den gewünschten Pegel annimmt, da dazu jedesmal der zugehörige Kondensator umgeladen werden muß.

Die Knöpfe der Eingabegeräte

Jedes der drei oben erwähnten Eingabegeräte besitzt einen oder mehrere Knöpfe. Folgende Tabelle zeigt, welche Register den Status der Knöpfe von Maus, Paddles und Joystick enthalten:

Gameport 0:

Linker Mausknopf	CIA-A, Parallelport A, Portbit 6
Rechter Mausknopf	POTGOR, DATLY
(Dritter Mausknopf	POTGOR, DATLX)
Joystick-Feuerknopf	CIA-A, Parallelport A, Portbit 6
Linker Paddel-Knopf	JOY0DAT, Bit 9 (1 = Knopf gedrückt)
Rechter Paddel-Knopf	JOY0DAT, Bit 1 (1 = Knopf gedrückt)

Gameport 1:

Linker Mausknopf	CIA-A, Parallelport A, Portbit 7
Rechter Mausknopf	POTGOR, DATRY
(Dritter Mausknopf	POTGOR, DATRX)
Joystick-Feuerknopf	CIA-A, Parallelport A, Portbit 7
Linker Paddel-Knopf	JOY1DAT, Bit 9 (1 = Knopf gedrückt)
Rechter Paddel-Knopf	JOY1DAT, Bit 1 (1 = Knopf gedrückt)

(Wenn nicht anders angegeben, sind die Bits Null-aktiv, d.h. 0 = Knopf gedrückt.)

1.5.10 Die serielle Schnittstelle

Wie man im Kapitel 1.3.4 nachlesen konnte, besitzt der Amiga eine standardmäßige RS232-Schnittstelle. Die verschiedenen Leitungen dieser Buchse kann man in zwei Signalgruppen einteilen:

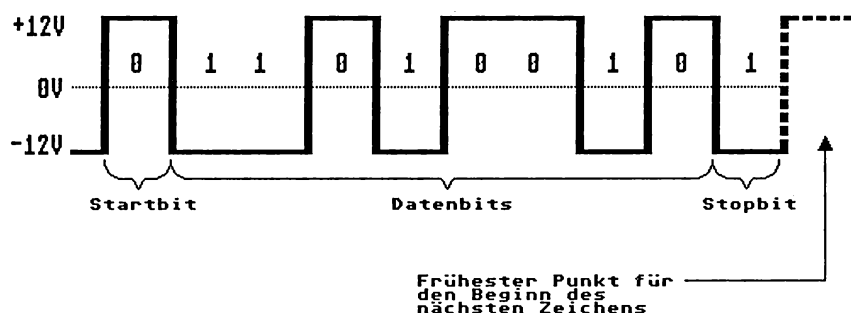
1. Die seriellen Datenleitungen
2. Die Handshake-Leitungen

Erst zu 2: Die RS232-Schnittstelle besitzt eine Vielzahl von Handshake-Leitungen. Meistens werden gar nicht alle davon gebraucht. Außerdem ist das Verhalten dieser Signale nicht bei jedem RS232-Gerät gleich. Ihre Funktionsweise und Programmierung beim Amiga wurden schon in 1.3.4 beschrieben.

Zu 1:

Über die beiden Datenleitungen läuft die gesamte Informationsübertragung. Die RXD-Leitung empfängt die Daten, und über TXD werden sie ausgegeben. Die RS232-Kommunikation kann daher in zwei Richtungen gleichzeitig ablaufen, wenn zwei Geräte über RXD und TXD miteinander verbunden sind. Man koppelt dabei RXD des einen Geräts mit TXD des anderen und umgekehrt.

Prinzip der seriellen RS232-Datenübertragung



Ablauf der seriellen Datenübertragung

Abb. 1.5.10

Da für die Datenübertragung pro Richtung nur eine einzige Leitung vorhanden ist, müssen die Datenworte in einen seriellen Datenstrom verwandelt werden, der dann Bit für Bit übertragen werden kann. Bei der RS232-Norm sind keinerlei Taktleitungen vorgesehen. Damit der Empfänger weiß, wann er das nächste Bit einlesen kann, muß die Zeit pro Bit konstant sein, d.h. die Geschwindigkeit, mit der die Daten ausgegeben und eingelesen werden, muß festgelegt werden. Dies geschieht durch die sogenannte Baudrate. Sie bestimmt die Anzahl der übertragenen Bits pro Sekunde. Gebräuchliche Baudraten sind beispielsweise 300,

1200, 2400, 4800 und 9600 Baud. Man muß sich aber nicht an diese Standardwerte halten, allerdings sollte bei "krummen" Baudraten beachtet werden, daß Sender und Empfänger auch wirklich übereinstimmen.

Damit die Übertragung klappt, muß der Empfänger noch wissen, wann ein Byte beginnt und endet. Die Abbildung 1.5.11 zeigt den zeitlichen Ablauf eines Bytes auf einer der Datenleitungen. Jedes Byte beginnt mit einem Startbit, das sich nicht von den normalen Datenbits unterscheidet, allerdings immer den Wert 0 hat. Darauf folgen die Datenbits in der Reihenfolge vom LSB zum MSB. Am Ende schließen sich noch ein oder zwei Stoppbits an, die den Wert 1 haben. Der Empfänger erkennt den Wechsel von einem Byte zum nächsten an dem Pegelwechsel von 1 auf 0, der bei der Aufeinanderfolge von Stoppbit und Startbit auftritt.

Der Baustein, der diese serielle Übertragung ausführt, heißt Universal Asynchronous Receive Transmit, abgekürzt UART. Er ist beim Amiga innerhalb von Paula untergebracht, und seine Register befinden sich im Registerbereich der Customchips:

Die UART-Register

SERPER \$032 (nur schreiben)

Bit Nr.	Name	Funktion
---------	------	----------

15	LONG	Länge der Empfangsdaten auf 9 Bit setzen
0 bis 14	RATE	Diese 15-Bit-Zahl enthält die Baudrate.

SERDAT \$030 (nur schreiben)

SERDAT enthält die Sendedaten.

SERDATR \$018 (nur lesen)

Bit Nr.	Name	Funktion
15	OVRUN	Überlauf des Empfangsschieberegisters
14	RBF	Empfangsdatenpuffer voll
13	TBE	Sendedatenpuffer leer
12	TSRE	Sendeschieberegister leer
11	RXD	Entspricht dem Pegel der RXD-Leitung
10	---	unbenutzt
9	STP	Stoppbit
8	STP o. DB8	Hängt von eingestellter Datenlänge ab
7	DB7	Empfangsdatenpuffer Datenbit Nr. 7
6	DB6	Empfangsdatenpuffer Datenbit Nr. 6
5	DB5	Empfangsdatenpuffer Datenbit Nr. 5
4	DB4	Empfangsdatenpuffer Datenbit Nr. 4
3	DB3	Empfangsdatenpuffer Datenbit Nr. 3
2	DB2	Empfangsdatenpuffer Datenbit Nr. 2
1	DB1	Empfangsdatenpuffer Datenbit Nr. 1
0	DB0	Empfangsdatenpuffer Datenbit Nr. 0

Ein Bit im ADKCON-Register gehört ebenfalls zur UART-Steuerung:

ADKCON \$09E (schreiben) ADKCONR \$010 (lesen)

Bit Nr. 11: UARTBRK

Dieses Bit unterbricht die serielle Ausgabe und setzt TXD auf 0.

Ablauf der Datenübertragung beim Amiga-UART

Empfangen

Der Empfang der seriellen Daten läuft zweistufig ab. Die am RXD-Pin ankommenden Bits werden im Takt der Baudrate in ein Schieberegister übernommen und dort wieder zu einem parallelen Datenwort zusammengesetzt. Ist das Schieberegister voll, wird sein Inhalt in den Empfangsdatenpuffer geschrieben. Damit ist es sofort wieder für die nächsten Daten frei. Der

Prozessor kann nur den Empfangsdatenpuffer, nicht aber das Schieberegister auslesen. Die entsprechenden Datenbits heißen DB0 bis DB7 oder DB8 im SERDATR-Register.

Der Amiga kann sowohl acht als auch neun Bit-Datenworte empfangen. Mittels des LONG-Bits im SERPER-Register läßt sich der UART auf 9 Datenbits umschalten. Man setzt dazu LONG = 1.

Die eingestellte Datenlänge bestimmt über das Format im SERDATR-Register. Bei 9 Bits enthält Bit 8 von SERDATR das 9. Datenbit, während sich das Stoppbit in Bit 9 befindet. Bei 8 Datenbits enthält schon Bit 8 das Stoppbit. Erst das zweite Stoppbit, wenn überhaupt vorhanden, landet in Bit 9.

Den Zustand des Empfangsschieberegisters und des Datenpuffers geben zwei Signalbits im SERDATR wieder:

RBF steht für Receive-Buffer-Full, d.h. sobald ein Datenwort aus dem Schieberegister in den Puffer übertragen wird, springt dieses Bit auf 1 und signalisiert damit dem Mikroprozessor, daß er die Daten aus SERDATR auslesen soll.

Dieses Bit existiert auch noch einmal in den Interrupt-Registern (RBF, INTREQ/INTEN Bit Nr. 11). Nachdem der Prozessor die Daten gelesen hat, muß er RBF in INTREQ zurücksetzen. Es geht dann sowohl in INTREQ als auch in SERDATR wieder auf 0.

```
MOVE.W #$0800,$DFF000+INTREQ ;Löscht RBF in INTREQ undSERDATR
```

Unterläßt man dies, und das Schieberegister hat ein weiteres komplettes Datenwort empfangen, setzt der UART das OVRUN-Bit. Dies signalisiert, daß jetzt keine weiteren Daten mehr angenommen werden können, da sowohl der Puffer (RBF = 1) als auch das Schieberegister (OVRUN = 1) voll sind. OVRUN geht auf 0, wenn RBF zurückgesetzt wird. RBF springt danach wieder auf 1, da der Inhalt des Schieberegisters sofort nach DB0 bis DB8 übertragen wird, um das Schieberegister für neue Daten frei zu machen.

Senden

Auch der Sendevorgang läuft zweistufig ab. Der Sendedatenpuffer befindet sich in dem SERDAT-Register. Sobald man ein Datenwort dort hineinschreibt, wird es in das Ausgabeschieberegister übertragen. Dies signalisiert das TBE-Bit. TBE steht für Transmit-Buffer-Empty und zeigt an, daß SERDAT zur Aufnahme der nächsten Daten bereit ist. Auch TBE ist noch einmal in den Interrupt-Registern vorhanden (TBE, INTREQ/INTEN Bit Nr. 0). Wie RBF muß auch TBF durch Löschen im INTREQ-Register zurückgesetzt werden.

Hat das Schieberegister das Datenwort ausgegeben, wird automatisch das nächste aus dem Sendedatenpuffer geholt. Ist dort nicht rechtzeitig ein neues hineingeschrieben worden, setzt der UART das TSRE-Bit (Transmit-Shift-Register-Empty / Sendeschieberegister leer) auf 1. Dieses Bit wird mit dem Löschen von TBE zurückgesetzt.

Die Länge des Datenworts und die Anzahl der Stoppbits werden durch das Format der Daten in SERDAT festgelegt. Man schreibt einfach das gewünschte Datenwort in die unteren 8 oder 9 Bits von SERDAT und setzt, je nach Anzahl der Stoppbits, ein oder zwei Einsen davor. Ein 8 Bit Datenwort mit zwei Stoppbits sähe beispielsweise so aus:

Bit Nr.:	15	14	13	12	11	9	8	7	6	5	4	3	2	1	0
Funktion:	0	0	0	0	0	1	1	D7	D6	D5	D4	D3	D2	D1	D0

D0 bis D7 sind die acht Datenbits.

Die beiden Einsen stehen für die gewünschten zwei Stoppbits.

Bei einem 9-Bit-Datenwort mit einem Stoppbit müßte man folgende Daten in SERDAT schreiben:

Bit Nr.:	15	14	13	12	11	9	8	7	6	5	4	3	2	1	0
Funktion:	0	0	0	0	0	1	D8	D7	D6	D5	D4	D3	D2	D1	D0

Acht Bits plus ein Stoppbit:

Bit Nr.:	15	14	13	12	11	9	8	7	6	5	4	3	2	1	0
Funktion:	0	0	0	0	0	0	1	D7	D6	D5	D4	D3	D2	D1	D0

Das LONG-Bit im SERPER-Register hat nur auf die Länge der Empfangsdaten einen Einfluß. Das Format der Sendedaten wird allein von dem Wert im SERDAT-Register bestimmt.

Festlegen der Baudrate

Die Baudrate muß für Sende- und Empfangsdaten gemeinsam in die unteren 15 Bits des SERPER-Registers geschrieben werden. Leider kann die Baudrate nicht direkt eingestellt werden. Man muß die Anzahl der Buszyklen wählen, die zwischen zwei Bits liegen sollen (1 Buszyklus dauert $2.79365 \cdot 10^{-7}$ Sekunden). Soll alle n Buszyklen ein Bit ausgegeben werden, so muß man den Wert n-1 in das SERPER-Register schreiben. Mit folgender Formel kann man aus der Baudrate den notwendigen SERPER-Wert berechnen:

$$\text{SERPER} = \frac{1}{\text{Baudrate} \cdot 2.79365 \cdot 10^{-7}} - 1$$

Z.B. für eine Baudrate von 4800 Baud:

$$\text{SERPER} = 1 / (4800 \cdot 2.79365 \cdot 10^{-7}) - 1 = 1 / 0.00134 - 1 = 744.74$$

Der errechnete Wert wird gerundet und in SERPER geschrieben:

```
MOVE.W #745,$DFF000+SERPER ;SERPER setzen, LONG = 0
```

```
bzw. MOVE.W #$8000+745,$DFF000+SERPER ;LONG = 1
```

1.5.11 Der Diskcontroller

Die Steuerung der Diskettenlaufwerke gliedert sich hardwaremäßig in zwei Teile.

Als erstes gibt es die Steuerleitungen, die das gewünschte Laufwerk aktivieren, den Motor einschalten, den Schreib-/Lesekopf bewegen usw. Sie führen alle zu bestimmten Port-Leitungen der CIAs. Ihre Funktion und Verschaltung kann man im Kapitel 1.3.5 nachlesen.

Ausgenommen hiervon sind die Datenleitungen. Über sie laufen die Daten vom Schreib-/Lesekopf in den Amiga und beim Schreiben in umgekehrter Richtung vom Amiga auf die Diskette. Ein spezieller Baustein innerhalb von Paula, der Diskcontroller, übernimmt die Verarbeitung der Daten.

Er besitzt einen eigenen DMA-Kanal und schreibt oder liest selbständig die Daten auf bzw. von der Diskette.

Die Programmierung des Disk-DMA

Bevor man einen Disk-DMA startet, muß man sich vergewissern, daß der letzte schon beendet ist. Unterbricht man einen laufenden Schreibzugriff, kann man die Daten auf der entsprechenden Spur zerstören. Setzen wir also voraus, der letzte Disk-DMA sei abgeschlossen.

Als erstes muß die Speicheradresse des Datenpuffers festgelegt werden. Der Disk-DMA verwendet eines der üblichen Adreßregisterpaare als Zeiger auf das Chip-RAM. Sie heißen DSKPTH und DKSPTL:

\$20 DSKPTH Zeiger auf Daten von/zur Diskette Bits 16-18

\$22 DSKPTL Zeiger auf Daten von/zur Diskette Bits 0-15

Als nächstes muß das DSKLEN-Register initialisiert werden. Es ist wie folgt aufgebaut:

DSKLEN \$024 (nur schreiben)

Bit Nr.	Name	Funktion
15	DMAEN	Disk-DMA einschalten
14	WRITE	Daten auf Diskette schreiben
0-13	LENGTH	Anzahl der zu übertragenden Datenworte

LENGTH Die unteren 14 Bits des DSKLEN-Registers enthalten die Anzahl der zu übertragenden Datenworte.

WRITE Mittels WRITE = 1 schaltet man den Diskcontroller von Lesen auf Schreiben um.

DMAEN Wenn man DMAEN auf 1 setzt, beginnt der Datentransfer. Allerdings muß man dabei einiges beachten:

1. Das Disk-DMA-Enable-Bit im DMACON-Register (DSKEN, Bit Nr. 4) muß ebenfalls gesetzt sein.

2. Um ein versehentliches Auslösen eines Schreibzugriffs auf Diskette zu erschweren, muß man zweimal hintereinander das DMAEN-Bit setzen. Erst danach beginnt der Disk-DMA. Außerdem sollte das WRITE-Bit aus Sicherheitsgründen nur während eines Schreibzugriffs auf 1 sein. Eine ordentliche Initialisierungssequenz für einen Disk-DMA-Zugriff sieht wie folgt aus:

1. DSKLEN mit 0 beschreiben, dies schaltet DMAEN aus.

2. Falls DSKEN im DMACON-Register noch nicht gesetzt ist, sollte man das an dieser Stelle erledigen.

3. Die gewünschte Adresse nach DSKPTH und DSKPTL.

4. Den richtigen Wert für LENGTH und WRITE zusammen mit gesetztem DMAEN-Bit nach DSKLEN schreiben.

5. Den gleichen Wert noch einmal in DSKLEN schreiben

6. Warten, bis Disk-DMA beendet (s.u.).
7. Danach DSKLEN sicherheitshalber auf 0 zurücksetzen.

Damit der Prozessor weiß, wann der Diskcontroller die in LENGTH festgelegte Anzahl von Worten übertragen hat, gibt es den sogenannten DSKBLK-Interrupt (Disk Block Finished, Bit Nr. 1 in INTREQ/INTEN). Er wird ausgelöst, nachdem das letzte Datenwort gelesen bzw. geschrieben wurde.

Den aktuellen Status des Diskcontrollers kann man im DSKBYTR-Register auslesen:

DSKBYTR \$01A (nur lesen)

Bit Nr.	Name	Funktion
15	BYTEREADY	Dieses Bit signalisiert, daß das Datenbyte in den unteren 8 Bits gültig ist.
14	DMAON	DMAON zeigt an, ob der Disk-DMA eingeschaltet ist. Damit DMAON = 1 ist, müssen sowohl DMAEN in DSKLEN und DSKEN in DMACON gesetzt sein.
13	DSKWRITE	Gibt den Zustand von WRITE in DSKLEN an.
12	WORDEQUAL	Diskdaten gleich DSKSYNC
11 bis 8		unbenutzt
7 bis 0	DATA	Aktuelles Datenbyte von Diskette

Mittels der 8 DATA-Bits und des BYTEREADY-Flags kann man die Daten von der Diskette statt per DMA mit dem Prozessor lesen. Jedesmal, wenn ein komplettes Datenbyte empfangen wurde, setzt der Diskcontroller das BYTEREADY-Bit. Damit weiß der Prozessor, daß das Datenbyte in den 8 DATA-Bits gültig ist. Nach dem Lesen des DSKBYTR-Registers wird BYTEREADY automatisch zurückgesetzt.

Manchmal kommt es vor, daß man nicht eine ganze Spur auf einmal in den Speicher lesen will. In diesem Fall besteht die Möglichkeit, den DMA-Transfer erst an einer ganz bestimmten Position zu starten. Dazu schreibt man das Datenwort, an dem der Diskcontroller beginnen soll, in das DSKSYNC-Register:

DSKSYNC \$07E (nur schreiben)

DSKSYNC enthält das Datenwort, an dem die Übertragung beginnen soll.

Der Diskcontroller fängt dann nach dem Einschalten des Disk-DMA's wie gewöhnlich damit an, die Daten von der Diskette zu lesen, er schreibt sie aber noch nicht in den Speicher. Statt dessen vergleicht er sie ständig mit dem Datenwort in DSKSYNC. Erst wenn beide übereinstimmen, beginnt er mit der Datenübertragung, die dann genauso wie sonst auch abläuft. Damit kann man den Diskcontroller so programmieren, daß er die Synchronisationsmarkierung am Anfang eines Datenblocks abwartet.

Das WORDEQUAL-Bit im DSKBYTER-Register ist 1, sobald gelesene Daten und DSKSYNC übereinstimmen. Da diese Übereinstimmung immer nur zwei (bzw. 4) Mikrosekunden dauert, ist WORDEQUAL auch nur innerhalb dieser Zeitspanne gesetzt. Zusätzlich wird jedesmal ein Interrupt erzeugt, wenn WORDEQUAL auf 1 geht:

Bit Nr. 12 im INTREQ- bzw. INTEN-Register ist das DSKSYN-Interruptbit. Es wird gesetzt, wenn die Daten von der Diskette mit DSKSYNC übereinstimmen.

Einstellen der Betriebsparameter

Die Daten können nicht in demselben Format auf die Diskette geschrieben werden, wie sie im Speicher stehen. Sie müssen speziell codiert werden. Normalerweise arbeitet der Amiga mit der sogenannten MFM-Codierung. Es ist aber auch möglich, die GCR-Codierung zu verwenden. Zwei Schritte sind zur Auswahl der gewünschten Codierung notwendig:

1. Eine entsprechende Routine muß die Daten vor dem Schreiben auf die Diskette codieren und die gelesenen Daten decodieren.
2. Der Diskcontroller muß auf die entsprechende Codierung eingestellt werden. Dies geschieht durch bestimmte Bits im ADKCON-Register.

ADKCON \$09E (schreiben) ADKCONR \$010 (lesen)

Bit Nr.	Name	Funktion															
15	SET/CLR	Bits setzen (SET/CLR=1) oder löschen															
14-13	PRECOMP	Diese Bits enthalten den Precomp-Wert: <table> <tr> <th>Bit 14</th><th>Bit 13</th><th>PRECOMP-Zeit</th></tr> <tr> <td>0</td><td>0</td><td>Null</td></tr> <tr> <td>0</td><td>1</td><td>140 Nanosekunden</td></tr> <tr> <td>1</td><td>0</td><td>280 Nanosekunden</td></tr> <tr> <td>1</td><td>1</td><td>560 Nanosekunden</td></tr> </table>	Bit 14	Bit 13	PRECOMP-Zeit	0	0	Null	0	1	140 Nanosekunden	1	0	280 Nanosekunden	1	1	560 Nanosekunden
Bit 14	Bit 13	PRECOMP-Zeit															
0	0	Null															
0	1	140 Nanosekunden															
1	0	280 Nanosekunden															
1	1	560 Nanosekunden															
12	MFMPREC	0 = GCR, 1 = MFM															
11	UARTBRK	kein Bit des Diskcontrollers, siehe UART															
10	WORDSYNC	WORDSYNC = 1 schaltet die oben beschriebene Synchronisation des Diskcontrollers nach dem Wort im DSKSYNC-Register ein.															
9	MSBSYNC	MSBSYNC = 1 schaltet auf GCR-Synchronisation															
8	FAST	Taktrate des Diskcontrollers: FAST=1: 2 Mikrosekunden/Bit (MFM) FAST=0: 4 Mikrosekunden/Bit (GCR)															
7-0	AUDIO	Diese Bits gehören nicht mehr zum Diskcontoller, siehe Kapitel 1.5.8															

Die Datenregister des Diskcontrollers

Wie üblich transportiert der DMA-Controller die Daten aus dem Speicher in die entsprechenden Datenregister. Der Diskcontroller besitzt ein Datenregister für die von Diskette gelesenen Daten und eines für die Daten, die auf die Diskette geschrieben werden sollen.

DSKDAT \$026 (nur schreiben)

Enthält die Daten, die auf die Diskette geschrieben werden.

DSKDAT \$008 (nur lesen)

Enthält die Daten von der Diskette. Dies ist ein sogenanntes Early-Read-Register und kann nicht vom Prozessor gelesen werden.

2. Exec

In diesem Kapitel wollen wir uns das Betriebssystem des Amiga ansehen, da es erfolgreichen Programmierer (oder dem, der es werden will) zumindest in groben Zügen bekannt sein muß.

2.1 Grundlagen des Betriebssystems

Das Betriebssystem, das beim Amiga 1000 von der Kickstart-Diskette geladen werden muß, liegt im obersten Adressierungsbereich und umfasst 256 KB (\$FC0000 - \$FFFFFF). In diesem großen Speicherbereich findet eine gewaltige Anzahl Routinen Platz, die dem Programmierer seine Arbeit enorm erleichtern. Diese Routinen sind beim Amiga nach den verschiedenen Aufgaben sortiert. Man kann sich das System aus einer Anzahl einzelner gut aufeinander abgestimmter Module vorstellen, die sich die verschiedenen Aufgaben teilen. Die wichtigsten Teile sind sicherlich: DOS (Ein-/Ausgabesteuerung), Grafik, Intuition (Ansammlung komplexer Routinen, die sich zum größten Teil auf die Window- und Screen-Verwaltung beziehen) und Exec.

Die Aufgabe von Exec ist es, das Multitasking zu verwalten und somit ein paralleles Arbeiten mehrerer Programme zu ermöglichen. Als weiteres stellt Exec die unterste Ebene zwischen der Hardware und dem Programm dar. Anhand dieser Aufgaben ist schon zu sehen, daß Exec der wichtigste Teil des Amiga-Betriebssystems ist.

Jedes Teilsystem stellt eine Anzahl leistungsfähiger Routinen zur Verfügung, die vom Programmierer leicht genutzt werden können. Damit die Routinen besser aufzurufen sind, werden die jeweils zu einem Betriebssystemteil gehörigen Routinen in einer Jump-Tabelle zusammengefaßt. Wie sich diese Routinen aufrufen lassen, wird in Kapitel 2.3 erläutert.

Die Kommunikation zwischen Exec und der Hardware läuft nicht direkt ab, sondern wird letztendlich von einem Device-Handler (Gerätetreiber) erledigt. Sämtliche Routinen zum An-

sprechen der Device-Handler werden von Exec zur Verfügung gestellt. Es ist natürlich auch möglich, die Hardware des Amiga ohne einen Device-Handler in Maschinensprache direkt anzusprechen, damit würde man jedoch auf die Multitasking-Fähigkeiten des Systems weitgehend verzichten müssen.

2.2 Einführung in die Programmierung des Amiga

Nachdem der Aufbau des Betriebssystems des Amiga grob erklärt wurde, wollen wir uns näher mit der eigentlichen Programmierung beschäftigen.

Ein Hinweis schon an dieser Stelle:

Sämtliche Assembler-Listings aus dem Kickstart-Bereich beziehen sich auf die Kickstartversion 1.2, die sowohl beim Amiga 500 als auch beim Amiga 2000 im ROM eingebaut ist. Für die Besitzer eines Amiga 1000 stimmen die Routinen nur überein, wenn sie die gleiche Kickstartversion benutzen.

2.2.1 Unterschiede bei C und Assembler

Daß Assembler um einiges schneller als C und deshalb für manche Probleme besser zu gebrauchen ist als C oder eine andere Hochsprache, dürfte jeder von Ihnen wissen. Was man jedoch bei dem Gebrauch von Betriebssystem-Routinen oder Strukturen beachten muß, ist wahrscheinlich weniger bekannt.

Fangen wir beim Aufrufen von Routinen mit Übergabeparametern an. In C sieht dies folgendermaßen aus:

```
Eintrag = FindName (list,"name");  
D0                A0    A1
```

FindName ist eine Routine zum Auffinden eines Eintrags in einer Liste von Einträgen mit Hilfe des Namens des Eintrags. Die Parameter, die übergeben werden müssen, sind Zeiger auf den Anfang der verketteten Liste und den Namen des Eintrags,

nach dem gesucht wird. Für Assembler-Programmierer stehen bei den von uns beschriebenen Routinen (Funktionen) die Register, in denen die Parameter übergeben werden müssen, wenn man die Funktion aufruft. Zurück erhält man den Zeiger auf den Eintrag, der in 'Eintrag' gespeichert wird. Der Zeiger, den man von der Funktion zurückerhält, wird bei dieser und bei anderen Funktionsaufrufen in Register D0 übergeben. Wenn es mehr als einen Rückgabeparameter gibt, werden auch andere Register mitbenutzt.

In Assembler ist der Aufruf prinzipiell gleich. Der Routine müssen die Parameter nur in den entsprechenden Registern übergeben werden. Dies sieht dann wie folgt aus:

```
LEA.L LIST,A0
LEA.L NAME,A1
JSR FINDNAME
MOVE.L D0,SPEICHER
:
NAME:
DC.B "NAME",0
```

Als erstes wird der Zeiger auf die Liste in A0 geschrieben. In A1 muß der Zeiger auf den Namen stehen der gesucht werden soll. Der String muß mit Null abgeschlossen werden. Daraufhin wird die Routine aufgerufen und der Zeiger auf den gefundenen Eintrag in D0 übergeben, von wo aus er zwischengespeichert werden kann. Es ist üblich, daß die Rückmeldung in D0 erfolgt. Letztendlich wird der in C geschriebene Aufruf auch so übersetzt.

Soviel zum Aufrufen von Routinen. Als nächstes sehen wir uns an, wie C-Strukturen in Assembler aussehen.

```
struct Node {
    struct Node *ln_Succ;
    struct Node *ln_Pred;
    UBYTE      ln_Type;
    BYTE       ln_Pri;
    char       *ln_Name;
};
```

Auf die Bedeutung der Struktur wollen wir an dieser Stelle noch nicht eingehen. Sie dient lediglich zur Demonstration der Unterschiede zwischen C und Assembler. Das Initialisieren der Struktur dürfte kein Problem darstellen.

Beispielsweise:

```
ln_Pri = 20;
```

Mit dieser Zuweisung wird der Wert 20 als ein Einbyte-Wert mit Vorzeichen in der Struktur vermerkt.

In Assembler besteht eine solche C-Struktur aus einer Tabelle. Die Werte sind in gleicher Reihenfolge, die in der Struktur angegeben ist, in ihrer zugehörigen Länge abgelegt. Für eine solche Tabelle muß ihre Basisadresse bekannt sein, damit man sie ansprechen kann.

Für dieses Beispiel sieht das wie folgt aus:

Basis + 0	\$00000000	Zeiger auf Nachfolger	ln_Succ
Basis + 4	\$00000000	Zeiger auf Vorgänger	ln_Pred
Basis + 8	\$00		ln_Type
Basis + 9	\$00		ln_Pri
Basis + 10	\$00000000	Zeiger auf Name	ln_Name

Die Nullen stehen für beliebige Werte. Um ln_Pri auf 20 zu setzen, wie wir es zuvor in C gemacht haben, muß die Basisadresse der Struktur bekannt sein. Ist dies der Fall, so steht dem Setzen des ln_Pri-Feldes nichts mehr im Wege.

```
LEA.L    Basis+9,A0
MOVE.B   #20,(A0)
```

Sie sehen, daß auch das Ansprechen von Strukturen aus Assembler kein Problem bedeutet. Natürlich ist es nicht so bequem wie mit C. Assembler hat jedoch den Vorteil, schneller zu sein als C, und man kann hardwareorientierter damit arbeiten. Mit Assembler lassen sich manche Routinen ansprechen, die aus C nicht erreichbar sind. Diese Routinen beziehen sich auf

Teile im Betriebssystem, auf die der "normale" Programmierer keinen Einfluß zu haben braucht, die aber für manche Tricks, z.B. bei der Multitaskingverwaltung, zu gebrauchen sind.

2.2.2 Aufbau von Nodes (Knoten)

Als nächstes wollen wir auf die wichtigsten Grundstrukturen eingehen, deren Verständnis unumgänglich ist, wenn die folgenden Kapitel verstanden werden sollen.

Als erstes wird die Node-Struktur vorgestellt. Sie wird benutzt, um verkettete Listen herzustellen, welche im Amiga sehr häufig zu finden sind. Diese Struktur ist wohl die meistbenutzte Struktur im Amiga-Betriebssystem. Sie hat folgendes Aussehen:

In C:

```
struct Node {
    struct Node *ln_Succ;
    struct Node *ln_Pred;
    UBYTE      ln_Type;
    BYTE       ln_Pri;
    char       *ln_Name;
};
```

In Assembler:

Basis + 0	\$00000000	Zeiger auf Nachfolger ln_Succ
Basis + 4	\$00000000	Zeiger auf Vorgänger ln_Pred
Basis + 8	\$00	ln_Type
Basis + 9	\$00	ln_Pri
Basis + 10	\$00000000	Zeiger auf Name ln_Name

Diese Struktur kann man in zwei Teile aufteilen. Zum einen besteht sie aus einem Verkettungsteil (ln_Succ und ln_Pred) und einem Datenteil (Type, Priority und Name).

**ln_Succ*

Dies ist ein Zeiger auf die nächste Node (Successor = Nachfolger).

**ln_Pred*

Das ist ein Zeiger auf die vorherige Node (Predecessor = Vorgänger).

ln_Type

In diesem Byte werden die verschiedenen Typen der Node entsprechend kodiert gespeichert.

ln_Pri

Hier wird die Priorität der Node vermerkt. Dieses Feld auf einen Wert ungleich Null zu setzen, ist nur in manchen Fällen, wie zum Beispiel bei einer Task-Node, sinnvoll. Doch dazu kommen wir später.

**ln_Name*

In diesem Langwort wird ein Zeiger auf einen mit Null abgeschlossenen String gespeichert. Es ist der Name der Node, der möglichst so gewählt werden sollte, daß man schon anhand des Namens erkennt, um welche Node es sich handelt, was die Fehlersuche enorm vereinfacht.

Initialisieren einer Node

Bevor eine Node in eine Liste eingehängt wird, sollte sie zuvor ordentlich initialisiert sein.

Zu diesem Zweck muß der Type gesetzt werden. Hierbei hat man eine Auswahl zwischen näheren standardisierten Typen, die folgend aufgeführt sind.

Node-Typ	Code
NT_TASK	01
NT_INTERRUPT	02
NT_DEVICE	03
NT_MSGPORT	04
NT_MESSAGE	05
NT_FREEMSG	06
NT_REPLYMSG	07
NT_RESOURCE	08
NT_LIBRARY	09
NT_MEMORY	10
NT_SOFTINT	11
NT_FONT	12
NT_PROCESS	13
NT_SEMAPHORE	14

Den Node-Typ anzugeben ist somit sehr leicht. Man sucht sich den zu seiner Node passenden Type aus der Tabelle und trägt ihn ein.

Nehmen wir einmal an, es soll die Node, die sich in einer Task-Struktur befindet, initialisiert werden. Um zu verstehen, wie diese Initialisierung dann aussieht, zeigen wir erst, wie eine Task-Struktur mit den für uns bis jetzt wichtigen Teilen aufgebaut ist.

```
struct Task {  
    struct Node tc_Node;  
    :  
    :  
};
```

Die Initialisierung des Node-Typs sieht in C also wie folgt aus:

```
struct Task meintask; /* meintask ist der Name der  
                       zugewiesenen TaskStruktur */  
meintask.tc_Node.ln_Type = NT_TASK;
```

Im Anschluß die gleiche Initialisierung noch in Assembler.

```
LEA.L meintask,A0      ;Basisadresse des Tasks nach A0
MOVE.L #01,8(A0)      ;Type = Task (Wert 01) setzen
```

Nachdem der Type festgelegt ist, wird die Priorität der Node im Vergleich zu den anderen Nodes angegeben. Sie kann einen Wert zwischen -128 und +127 annehmen. Ein größerer positiver Wert steht für eine höhere Priorität, somit ist +127 die höchste und -128 die niedrigste Priorität.

Einige Exec-Listen werden nach der Höhe der Priorität der sich in ihr befindlichen Einträge geordnet, wobei der Eintrag mit der höchsten Priorität am Anfang steht. Die meisten Exec benutzen den `ln_Pri`-Eintrag nicht. Es ist sicherlich sinnvoll, die Priorität solcher Listeneinträge (Nodes) auf Null zu setzen.

Das Setzen der Priorität geschieht wie folgt:

```
meintask.tc_Node.in_Pri = 5;
```

Und in Assembler:

```
LEA.L meintask,A0      ;Basisadresse des Tasks nach A0
MOVE.L #05,9(A0)      ;Pri auf 5 setzen
```

Zum Schluß müssen wir nur noch den Namen der Node und somit auch des Tasks angeben.

In C:

```
meintask.tc_Node.in_Name = "Beispiel Task";
```

In Assembler:

```
LEA.L meintask,A0      ;Basisadresse des Tasks nach A0
LEA.L Name,A1          ;Adresse an der der Name steht nach A1
MOVE.L A1,10(A0)       ;Zeiger auf Namen eintragen
Name:
DC.B "Beispiel Task",0
```

Der String muß, wie schon gesagt, mit Null abgeschlossen werden.

Anhand dieser Beispiele sehen Sie, daß Sie lediglich die Position des entsprechenden Eintrags wissen müssen, um die Struktur aus Assembler initialisieren zu können. Sie brauchen also einen sogenannten "Offset" (zu deutsch Ausgleich), von der Basisadresse ausgehend, um die richtige Position zu erreichen. Dieser Offset ist in unserem letzten Beispiel 10.

Das Initialisieren von `ln_Succ` und `ln_Pred` wird im nächsten Kapitel besprochen.

2.2.3 Listen

Was ist eigentlich eine Liste, und was steht in ihr drin? Eine Liste ist eine Reihe von Node-Strukturen, die vor- und rückwärts miteinander verkettet sind. Wie schon gesagt, steht an erster Stelle der Node-Struktur ein Zeiger (`ln_Succ`) auf die nächste Node. An zweiter Position ist ein Zeiger (`ln_Pred`) auf die vorausgegangene Node.

Um eine verkettete Liste besser verwalten zu können, hat man noch einen Listen-Kopf eingeführt, mit dem sich sofort der Anfang oder das Ende der verketteten Liste finden läßt. Abgesehen von dieser Information, ist in der Struktur auch eingetragen, um welche Art von Einträgen es sich handelt. Die List-Struktur hat folgendes Aussehen:

Die Zahlen vor den Strukturgliedern in der folgend aufgeführten Listenstruktur sind deren Offsets, damit man auch mit Assembler auf sie zugreifen kann. Die Offsets gehören nicht zur C-Struktur, und dürfen folglich nicht mit eingegeben werden. Sie dienen nur als Gedankenstütze für Assemblerprogrammierer.

```
struct List {  
0    struct Node *lh_Head;  
4    struct Node *lh_Tail;  
8    struct Node *lh_TailPred;  
12   UBYTE      lh_Type;  
13   UBYTE      lh_pad;  
};
```

**lh_Head*

ist ein Zeiger auf den ersten Eintrag (erste Node) der Liste.

**lh_Tail*

steht immer auf Null.

**lh_TailPred*

ist ein Zeiger auf den letzten gültigen Eintrag in der Liste.

**lh_Type*

gibt an, welche Sorte von Nodes in der Liste verkettet sind. lh_Type wird genau so gesetzt wie der Type der Nodes.

**lh_pad*

ist zwar in der Struktur enthalten, wird jedoch nicht verwendet.

Der Zeiger *ln_Succ* des letzten Eintrags der Liste zeigt auf *lh_Tail* (zweiter Eintrag der List-Struktur). *lh_Tail* ist Null (NIL) und zeigt dadurch an, daß die "vermeintlich" gelesene Node (es handelt sich jedoch in diesem Fall um die List-Struktur) nicht mehr gültig ist.

Der *ln_Pred*-Zeiger der ersten Node zeigt auf *lh_Succ* (also auch auf die List-Struktur). Wird nun der "vermeintliche" *n_Pred*-Zeiger gelesen, so ist dieser Null (es handelt sich ja hier um *lh_Tail*), was wiederum anzeigt, daß der zuvor gelesene Eintrag der Liste der erste war.

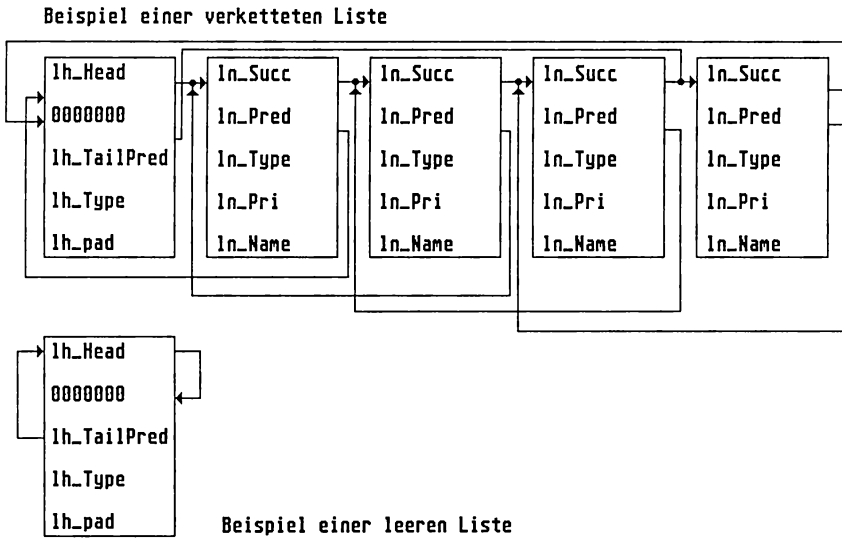


Abb. 2.2 Verkettete und leere Liste

Initialisierung einer Liste

Nachdem wir uns angesehen haben, wie eine verkettete Liste aussieht, wollen wir sie jetzt erstellen. Zu diesem Zweck muß zuerst eine neue List-Struktur erzeugt und als leer gekennzeichnet werden. Das Initialisieren einer Task-Liste sieht in C wie folgt aus:

```
#include "exec/lists.h"
main ()
{
    struct List liste;
    liste.lh_Head = (struct Node *)&liste.lh_Tail;
    liste.lh_Tail = 0;
    liste.lh_TailPred = (struct Node *)&liste.lh_Head;
    liste.lh_Type = NT_TASK;
}
```

In Assembler hätte es dieses Aussehen:

```
LEA.L    liste,A0
MOVE.L   A0,(A0)
ADDQ.L   #04,(A0)
CLR.L    #04(A0)
MOVE.L   A0,8(A0)
MOVE.B   #01,12(A0)
```

Die soeben erzeugte Liste ist selbstverständlich leer. Das Einfügen von Nodes in die Liste wird zu einem späteren Zeitpunkt besprochen. An dieser Stelle soll jedoch gezeigt werden, wie eine leere Liste auch als solche erkannt wird. Zur Feststellung gibt es zwei Methoden. Zum einen kann man untersuchen, ob `lh_Head` auf Null (NIL) zeigt, zum anderen, ob `lh_TailPred` auf den Anfang der Liste (`lh_Head`) zeigt. Ist das der Fall, so ist die betreffende Liste leer.

Die Abfrage sieht in C wie folgt aus:

```
if (liste.lh_TailPred == &liste) {
    printf ("liste ist leer");
}
```

Oder die andere Möglichkeit:

```
if (liste.lh_Head -> ln_Succ == 0) {
    printf ("liste ist leer");
}
```

2.2.4 Exec-Routine zur Listenverwaltung

Zur Verwaltung von Listen stellt Exec eine Reihe sehr nützlicher Funktionen zur Verfügung, mit denen sich nahezu alle sinnvollen Operationen bequem durchführen lassen.

Die erste Funktion ist die 'Insert()'-Funktion. Sie dient zum Einfügen von Nodes in eine Liste mit Angabe der Position, an der die neue Node eingefügt werden soll.

Insert

```
Insert (Liste,Node,Vorgänger);  
      A0   A1   A2
```

Offset: -234

Parameter:

Liste	ist ein Zeiger auf die Liste, in die die Node eingefügt werden soll.
Node	ist ein Zeiger auf die Node, die in die Liste eingefügt werden soll.
Vorgänger	ist ein Zeiger auf die Node, nach der die einzufügende Node eingefügt wird. Ist dieser Zeiger auf einen Wert ungleich Null gesetzt, so ist der Zeiger, der im Parameter 'Liste' gesetzt ist, nicht mehr relevant. Die angegebene Liste wird nicht nach der als Position angegebenen Node durchsucht (um festzustellen, ob die gewünschte Node überhaupt vorhanden ist), sondern setzt den Vorgänger Parameter als richtig voraus und fügt die Node ein. Steht der Parameter auf Null, so wird die Node an erster Position eingefügt. Die zweite Möglichkeit, eine Node an erster Position einzufügen, ist, 'Vorgänger' auf 'lh_Head' zu setzen. Ein Einfügen an letzter Position erreicht man durch Setzen des Vorgänger-Parameters auf 'lh_TailPred'. Für das Einbinden einer Node an erster oder letzter Stelle gibt es jedoch eigene Funktionen.

Remove**Remove (Node)****A1****Offset:** -252

Beschreibung: Wie aus dem Namen schon ersichtlich, dient diese Funktion zur Entfernung von Nodes aus einer Liste.

Parameter:

Node Zeiger auf die Node, die entfernt werden soll. Sollte es sich bei dem Zeiger nicht um einen Zeiger auf eine Node handeln, erkennt Exec dies nicht und "entfernt" trotzdem, wodurch es zum Informationsverlust oder zum "Absturz" des Rechners kommen kann.

AddHead**AddHead(liste,Node)****A0 A1****Offset:** -240

Beschreibung: Die Funktion wird benutzt, um Nodes am den Kopf einer Liste einzufügen.

Parameter:

Liste ist ein Zeiger auf die Liste, in die die Node eingefügt werden soll.

Node ist ein Zeiger auf die Node, die eingefügt werden soll.

RemHead

RemHead (Liste)
A0

Offset: -258

Beschreibung: Entfernt die erste Node der Liste, die angegeben wird.

Parameter:

Liste ist ein Zeiger auf die Liste, aus der die erste Node entfernt wird.

AddTail

AddTail (Liste,Node)
A0 A1

Offset: -246

Beschreibung: Einfügen einer Node als letztes Glied in der Liste.

Parameter:

Liste ist ein Zeiger auf die Liste, in die eingefügt werden soll.

Node ist ein Zeiger auf die Node, die eingefügt werden soll.

RemTail

RemTail (Liste)
A0

Offset: -258

Beschreibung: RemTail entfernt den letzten Eintrag der Liste.

Parameter:

Liste ist ein Zeiger auf die Liste, aus der der letzte Eintrag entfernt werden soll.

Enqueue

```
Enqueue (Liste,Node)
          A0   A1
```

Offset: -270

Beschreibung: Die Funktion wird verwendet, um Einträge in der Liste nach ihrer Priorität zu ordnen. Wie schon im Kapitel über Nodes gesagt, werden Nodes mit höherer Priorität an den Anfang der Liste gestellt. Sollten mehrere Nodes mit der gleichen Priorität in einer Liste vorhanden sein, so wird die neueingefügte Node hinter die vorhandenen gestellt.

Parameter:

Liste ist ein Zeiger auf die Liste, in die die Node eingetragen werden soll.

Node ist ein Zeiger auf die Node, die eingefügt werden soll.

FindName

```
Eintrag = FindName (Liste,"Name")
          D0           A0   A1
```

Offset: -276

Beschreibung: Mit 'FindName' kann man eine angegebene Liste nach einer Node mit entsprechendem Namen suchen.

Parameter:

Liste ist ein Zeiger auf die Liste, die durchsucht werden soll.

Name ist ein Zeiger auf den Namen, nach dem gesucht wird. Dieser String muß mit Null abgeschlossen werden. Beim Aufruf der C-Funktion handelt es sich bei dem Parameter nicht um einen Zeiger auf den Namen, sondern um den String selbst.

Ergebnis:

In 'Eintrag' (D0) wird von der Funktion ein Zeiger zurückgegeben, der auf die gefundene Node zeigt. Sollte ein Eintrag mit entsprechenden Namen nicht gefunden werden, wird eine Null zurückgegeben.

Das folgende Beispiel zeigt, wie man feststellen kann, ob ein Name einer Node doppelt in einer Liste vorkommt. Das Beispiel kann in der Form noch nicht gestartet werden, die Liste muß vor dem 'FindName()'-Aufruf initialisiert werden, da es sonst zu einem Absturz des Rechners kommt.

```
#include <exec/lists.h>

main()
{
    struct Node *FindName(),*node;
    struct List *liste;

    if ((node = FindName (liste,"testnode"))!=0)
        if((node = FindName (node,"testnode"))!=0)
            printf ("\n der Name testnode wurde 2 mal
            gefunden\n");
}
```

Nach der Auflistung der für die Bearbeitung der Liste zur Verfügung stehenden Funktionen sollen diese teilweise anhand eines Beispiels verdeutlicht werden.

Unser Beispiel zeigt, wie man eine Liste erstellt, ausgibt und ihre Einträge löscht.

Beispiel für Listen:

```

#include <exec/lists.h>

char *name[] = {"node1","node2","node3"};

struct List liste;
struct Node node[3],*np;

main ()
{
    int i;
    char n;
    liste.lh_Head = (struct Node *) &liste.lh_Tail;
    liste.lh_Tail = 0;
    liste.lh_TailPred = (struct Node *) &liste.lh_Head;
    liste.lh_Type = NT_TASK;

    for (i=0;i<=2;i++) {
        node[i].ln_Type = NT_TASK;
        node[i].ln_Name = name[i];

        AddTail (&liste,&node[i]);
    }

    ausgabe();
    printf ("\n Ausgabe der fertigen Liste\n");

    np = liste.lh_Head->ln_Succ;
    Remove (np);
    ausgabe();
    printf ("\n 2. Node wurde entfernt\n");
}

ausgabe ()
{
    for (np = liste.lh_Head;
        np != &liste.lh_Tail;
        np = np->ln_Succ)
        printf ("\n %s \n",np->ln_Name);
}

```

2.3 Funktionstabellen (Libraries)

Dieses Kapitel ist sowohl für C- als auch für Assembler-Programmierer gedacht. Das Verständnis des Kapitels ist unerlässlich, wenn man den Amiga mit all seinen Möglichkeiten nutzen will.

Was ist eigentlich eine Library? Eine Library ist laut Übersetzung eine Bibliothek. Es handelt sich hierbei um eine Bibliothek von Funktionen, die vom Programmierer genutzt werden können. Sie ist also, um genau zu sein, eine größere Sprungtabelle, über die die Funktionen aufgerufen werden. Libraries werden gebraucht, damit der Programmierer die Funktionen benutzen kann, die nicht in C und erst recht nicht in Assembler vorhanden sind, jedoch vom Betriebssystem zur Verfügung gestellt werden. Ein Beispiel dafür ist die Funktion 'OpenScreen()' zum Erstellen eines eigenen Screens. C stellt diese Funktion nicht zur Verfügung, sie muß mit Hilfe einer Library des Amiga aufgerufen werden.

Eine Library hat folgendes Aussehen:

```
      :  
      :  
00060A JMP $FC2FD6  
000610 JMP $FC0B28  
000616 JMP $FC0AC0  
      :  
      :
```

Die Libraries sind in verschiedene Aufgabenteile unterteilt. Die meisten von ihnen sind schon im ROM-Bereich des Rechners enthalten; einige wenige müssen bei Bedarf von Diskette nachgeladen werden. Diese Libraries stehen zur Verfügung:

```
clist.lib  
console.lib  
diskfont.lib  
dos.lib  
expansion.lib  
exec.lib  
graphics.lib
```

icon.lib
intuition.lib
layers.lib
mathffp.lib
mathieedoubbas.lib
mathtrans.lib
potgo.lib
ram.lib
timer.lib
translator.lib

Die Exec-Library nimmt eine Sonderstellung ein, denn ihre Funktionen sind sofort nach einem Reset erreichbar. Wenn man eine Funktion einer anderen Library benutzen will, muß man dies dem System mitteilen, welches daraufhin die entsprechende Library erstellt, auf die man dann zugreifen kann. Sollte diese Library schon erstellt worden sein, wird dem Programmierer lediglich mitgeteilt, wo er sie erreichen kann.

Von einer Library ist normalerweise lediglich ihre Basisadresse bekannt, von der aus mit negativen Offsets (Ausgleich) die gewünschten Funktionen aufgerufen werden.

Ein Library-Aufruf sieht, für alle Libraries geltend, wie folgt aus:

```
move.l libBasis,a6 ;Basisadresse der Library  
jsr    Offset(a6) ;Aufruf der Funktion mit negativem Offset
```

Der Offset der entsprechenden Funktion muß aus einer Tabelle entnommen werden, die Sie im Anhang dieses Buches finden. Vor dem Funktionsaufruf müssen die Parameter (sofern welche benötigt werden) den entsprechenden Registern übergeben werden.

Wie schon gesagt, sind die Funktionen der Exec-Library schon gleich nach einem Reset erreichbar. Um ihre Funktionen aufrufen zu können, muß ihre Basisadresse bekannt sein. Die Basisadresse der Exec-Library steht ab Speicherstelle \$04, von wo aus

sie ohne Probleme ausgelesen werden kann. Von C aus erreicht man sie durch Lesen der Standardvariablen 'SysBase'.

Ein Aufruf einer Exec-Library-Funktion aus C heraus ist sehr einfach. Als Beispiel zeigen wir den Aufruf der 'FindName'-Funktion.

```
#include <exec/execbase.h>
struct ExecBase *SysBase;
struct Library *FindName(), *library;

main()

{
    library = FindName(&(SysBase->LibList),"dos.library");
    printf ("\n %x \n",library);
}
```

Dieses kleine C-Programm durchsucht die Liste aller zur Verfügung stehenden Libraries nach der DOS-Library. Wird diese gefunden, so wird die Adresse, an der sie liegt, ausgegeben, ansonsten wird eine Null ausgegeben.

Sie sehen, daß es außer der für Sie vielleicht etwas verwirrenden Zuweisung innerhalb des Funktionsaufrufes bei Exec-Funktionen nichts zu beachten gibt. Sie wissen aus dem vorherigen Kapitel, daß der FindName-Funktion ein Zeiger auf eine Liste und der Name der Node mitgeteilt werden muß. Warum die Zuweisung der Liste dieses Aussehen hat, braucht hier noch nicht zu interessieren. Dies wird im Kapitel über 'ExecBase' erläutert.

Doch sehen wir uns einmal an, wie der C-Compiler dieses Programm in Assembler umsetzt. Wenn man sich auf das Wesentliche beschränkt, hat das umgesetzte Programm dieses Aussehen:

```

global _SysBase,4
global _library,4

_main:

    pea *Node           ;Zeiger auf Node in Stapel schieben
    move.l _SysBase,a6   ;Zeiger auf ExecBase (aus $4)
    pea 378(a6)          ;Zeiger auf Liste in Stapel schieben
    jsr _FindName        ;FindName Funktion aufrufen
    move.l d0,_library   ;Rückmeldung in library merken
    move.l _library,-(a7) ;und im Stapel an die
    jsr _printf          ;_printf Routine übergeben
    rts

_FindName
    movem.l 4(sp),a0/a1 ;Parameter für FindName aus Stapel in
                        ;die entsprechenden Register laden
    move.l _SysBase,a6 ;Basisadresse der Exec-Library holen
    jmp     -276(a6)    ;Funktion FindName aufrufen

```

Das komplette vom Compiler erzeugte Assemblerlisting ist etwas umfangreicher, trägt jedoch nicht zum Verständnis des Library-Funktionsaufrufs bei und wurde deshalb weggelassen.

`_SysBase` ist ein Zeiger auf die Exec-Library, dessen Funktionen mit negativen Offsets aufgerufen werden. Wenn man mit positiven Offsets von `_SysBase` aus auf den Speicher zugreift, erhält man Werte von ExecBase (der Hauptstruktur des Betriebssystems), auf die wir zu einem späteren Zeitpunkt noch eingehen werden. Unter den Einträgen in der ExecBase befindet sich auch eine Listenstruktur, in der die Libraries verzeichnet sind.

Das Compilat ist zwar ein wenig umständlich, aber trotzdem recht gut zu verstehen.

Das Hauptprogramm bringt die Zeiger auf die Liste und die Node in den Stapel und ruft daraufhin das Unterprogramm auf. Dort werden die eben gespeicherten Parameter in die dafür vorgesehenen Register geladen. Nun wird die Basisadresse der Exec-Library geholt und in A6 geschrieben. Dann ist alles vorbereitet, um die eigentliche Funktion 'FindName' aus der Exec-Library aufzurufen. Der Offset dieser Funktion ist, wie sich aus

dessen Beschreibung in Kapitel 2.2.4 entnehmen läßt, -276. Folglich heißt der eigentliche Funktionsaufruf 'jmp -276(a6)'. Der zurückgegebene Parameter wird in 'library' gespeichert, in den Stapel geschoben und mit '__printf' auf dem Bildschirm ausgegeben.

Wie schon gesagt, ist dem System der Standort der Exec-Library bekannt, weshalb ein Funktionsaufruf ohne Probleme geschehen kann. Sollte man eine Funktion aus einer anderen Library aufrufen wollen, so weiß weder das Betriebssystem noch der C-Compiler, wo er diese Library zu finden hat. Zu diesem Zweck stellt die Exec-Library eine Funktion zur Verfügung, mit der sich die Basisadressen anderer Libraries ermitteln lassen. Diese Funktion nennt sich 'OpenLibrary' und hat folgendes Aussehen:

```
LibPtr = OpenLibrary (LibName, Version)
D0                A1                D0
```

LibName

ist ein Zeiger auf den mit Null abgeschlossenen Namen der Library, die geöffnet werden soll, z.B. 'intuition.library'

Version

gibt die Library-Version an, die der Benutzer zu öffnen wünscht. Sollten mehrere Libraries desselben Namens zur Verfügung stehen, werden sie anhand der Versionen unterschieden. Sollte eine neue Version verlangt werden, die noch nicht zur Verfügung steht, so ist der 'OpneLibrary' Aufruf gescheitert.

LibPtr

beinhaltet nach dem Aufruf der Funktion die Basisadresse der gesuchten Library, sofern diese gefunden wurde. Ist die gesuchte Library nicht gefunden worden, wird eine Null als Fehlermeldung zurückgegeben.

Der Variablen-Name, in der die Basisadresse der Library gespeichert wird, kann nicht beliebig gewählt werden. Damit C

die Basisadresse der Library weiß, aus der eine Funktion aufgerufen werden soll, muß diese zuvor in einer dafür vorgesehenen Variable gespeichert werden. Wird diese Variable zwar deklariert, jedoch nicht auf den richtigen Wert gesetzt, führt der Aufruf der Library-Funktion zum Absturz des Programms oder sogar des ganzen Rechners.

Auflistung der festgelegten Variablen:

Library	Variable
clist.library	CListBase
diskfont.library	DiskFontBase
dos.library	DosBase
excansion.library	ExpansionBase
exec.library	SysBase
graphics.library	GfxBase
icon.library	IconBase
intuition.library	IntuitionBase
layers.library	LayersBase
mathffp.library	MathBase
mathieeedoubbas.library	MathIeeeDoubBasBase
mathtrans.library	MathtransBase
potgo.library	PotgoBase
translator.library	TranslatorBase

Es gibt mehrere Möglichkeiten, diese festen Variablen zu deklarieren. Am einfachsten ist es jedoch, sie als 'ULONG' (Langwort ohne Vorzeichen) zu deklarieren, weil man sich so die Pointerumwandlung erspart. Natürlich muß bei der Deklaration als 'ULONG' das Include-File 'exec/types.h' hinzugenommen werden, da der Compiler sonst nicht weiß, was 'ULONG' bedeutet.

2.3.1 Öffnen einer Library

Sehen wir uns den Gebrauch und das Öffnen einer Library einmal anhand eines Beispiels an. Das folgende Beispiel öffnet die Intuition-Library und daraufhin einen eigenen Screen.

```
#include <exec/types.h>
#include <intuition/intuition.h>

struct NewScreen ns = {
    0,0,
    640,256,
    2,
    0,1,
    HIRES,
    CUSTOMSCREEN,
    NULL,
    (UBYTE *) "My Screen",
    NULL,
    NULL };

ULONG IntuitionBase;

main()
{
    ULONG screen;

    if(!(IntuitionBase=OpenLibrary("intuition.library")))
        exit(100);

    screen = OpenScreen (&ns);
}
```

Nachdem `exec/types.h` und die Intuition-Strukturen eingebunden sind, wird die Screen-Struktur initialisiert und `IntuitionBase` als `ULONG` deklariert. `IntuitionBase` muß global deklariert werden, da sie in den von C eingebundenen Prozeduren (`OpenScreen`) verwendet werden muß. Dann wird die `'OpenLibrary()'`-Funktion der Exec-Library aufgerufen und die Intuition-Library geöffnet. In `IntuitionBase` wird die Basisadresse der Library gespeichert. Daraufhin wird die `'OpenScreen'` Funktion aufgerufen.

Sehen wir uns wieder an, was der Compiler aus diesem Programm macht. Das eigentliche Compilerprogramm ist etwas umfangreicher und wurde hier bis auf die wichtigsten Teile verkürzt.

```

_ns:  ;Initialisierung der Screen-Stuktur
dc.w 0
dc.w 0
dc.w 640
dc.w 256
dc.w 2
dc.b 0
dc.b 1
dc.w -32768
dc.w 15
dc.l $0000
dc.l .1+0
dc.l $0000
dc.l $0000
.1:
dc.b "My Screen",0

global _IntuitionBase,4

_main:

movem.l version,-(sp)      ;Lib. Version in den Stapel
pea *name                  ;Zeiger auf name in Stapel
jsr _OpenLibrary           ;OpenLib-Funktion aufrufen
move.l d0,_IntuitionBase  ;Rückmeldung speichern
tst.l d0                   ;auf Null testen
bne .5                     ;nicht Null (ok)
pea 100                    ;Nummer bei Exit
jsr _exit                  ;exit

.5:

pea _ns                    ;Zeiger auf Screen-Struktur
jsr _OpenScreen            ;Open-Screen Funktion aufrufen
rts                        ;Rücksprung

```

_OpenScreen:

```
move.l 4(sp),a0      ;Zeiger auf Screen-Struktur
move.l _IntuitionBase,a6 ;Basisadresse aus der festen
                        ;Variablen IntuitionBase holen
jmp     -198(a6)      ;Funktion ausführen
```

_OpenLibrary:

```
move.l _Sysbase,a6    ;Basisadresse von Exec holen
move.l 4(sp),a1        ;Zeiger auf Namen holen
move.l 8(sp),d0        ;Version aus Stapel holen
jmp     -552(a6)       ;OpenLibrary ausführen
```

Hier können Sie sehen, daß ohne die Deklaration von `IntuitionBase` das Programm nicht compiliert werden kann, da diese Variable in 'OpenScreen' verwendet wird. Sollte sie deklariert, jedoch nicht richtig (mit 'OpenLibrary') initialisiert worden sein, stürzt das Programm ab!

2.3.2 Schließen einer Library

Eine Library sollte immer geschlossen werden, wenn sie nicht mehr gebraucht wird, damit das Betriebssystem diese Library entfernen kann und mehr Speicher zur Verfügung steht.

Eine solche Funktion stellt die Exec-Library zur Verfügung. Sie lautet:

```
CloseLibrary (library)
              A1
```

library

ist ein Zeiger auf die offene Library, die geschlossen werden soll.

2.3.3 Struktur der Library

Die Library Funktion ist im C-Includefile wie folgt festgelegt:

```
#define LIB_VECTSIZE      6L
#define LIB_RESERVED      4L
#define LIB_BASE          (-LIB_VECTSIZE)
#define LIB_USERDEF       (LIB_BASE-
                           (LIB_RESERVED*LIB_VECTSIZE))
#define LIB_NONSTD        (LIB_USERDEF)

#define LIB_OPEN          (-6L)
#define LIB_CLOSE         (-12L)
#define LIB_EXPUNGE       (-18L)
#define LIB_EXTFUNC       (-24L)

#define LIBF_SUMMING (1L<<0)
#define LIBF_CHANGED (1L<<1)
#define LIBF_SUMUSED (1L<<2)
#define LIBF_DELEXP (1L<<3)

extern struct Library {
    0    struct  Node lib_Node;
    14   UBYTE   lib_Flags;
    15   UBYTE   lib_pad;
    16   UWORD   lib_NegSize;
    18   UWORD   lib_PosSize;
    20   UWORD   lib_Version;
    22   UWORD   lib_Revision;
    24   APTR    lib_IdString;
    28   ULONG   lib_Sum;
    32   UWORD   lib_OpenCnt;
};
```

Erklärung der Einträge der Struktur:

Lib_Node

ist eine Struktur der Sorte Node, wie wir sie schon kennengelernt haben. Mit Hilfe dieser Struktur sind die Libraries in einer Liste verkettet. Der Typ der Library ist in diesem Fall natürlich 'NT_LIBRARY' und der Name der Node gibt den Namen der Library an.

lib_pad

ist ein unbenutztes Byte, um die folgenden Worte und Langworte der Struktur wieder auf gerade Adressen zu bringen.

lib_NegSize

gibt an, wie groß der Bereich für die negativen Offsets ist.

Lib_PosSize

gibt an, wie groß der Bereich der Library von der Basisadresse an ist. Dieser Wert ist interessant, da eine Library mehr Einträge beinhalten kann, als in der C-Struktur vermerkt sind. Die Anzahl dieser Einträge und deren Bedeutung ist library-abhängig.

Lib_Version

gibt an, um welche Library-Version es sich handelt.

Lib_Revision

gibt an, um die wievielte Überarbeitung der Library es sich handelt.

Lib_IdString

ist ein Zeiger auf einen Text, der mehr Informationen über die Library enthält.

Lib_Summ

ist eine Prüfsumme über die Library. Wenn Sie die Library ändern, sollten Sie die Prüfsumme neu berechnen.

Lib_OpenCnt

gibt an, von wievielen Tasks die Library geöffnet ist. Es kommt auf den Typ der Library an, ob sie entfernt wird, wenn kein Task auf diese Library zugreifen will.

Die Funktionen, die eine Library dem Benutzer zu Verfügung stellt, beginnen mit Offset -30, obwohl sie theoretisch schon bei -6 beginnen könnten. Wenn man sich die ersten Offsets einmal genauer ansieht, stellt man fest, daß sie auf Funktionen zeigen, die von Exec benutzt werden, um die Library zu verwalten.

Hierbei handelt es sich um Funktionen, die zum Öffnen und Schließen der Library gebraucht werden und von den entsprechenden Exec-Funktionen angesprungen werden. Jede Library hat somit ihre eigene Öffnungs- und Schließroutine. In diesen eigenen Routinen wird auch entschieden, ob die Library, wenn sie von keinem Task mehr benötigt wird, entfernt werden darf oder nicht.

Wie auch aus den defines des Includefiles erkennbar ist, haben die besagten vier Offsets folgende Bedeutung:

LIB_OPEN	-6	Library öffnen
LIB_CLOSE	-12	Library schließen
LIB_EXPUNGE	-18	Library entfernen
LIB_EXTFUNC	-24	offen für Erweiterungen

Die Libraries, die nicht entfernt werden, wenn sie zur Zeit nicht mehr benötigt werden, benutzen den Einsprung LIB_EXPUNGE nicht. Alle nicht verwendeten Einsprünge sollten auf eine Routine zeigen, die D0 löscht und daraufhin wieder zurückkommt.

2.3.4 Ändern einer bestehenden Library

Zum Ändern einer bestehenden Library existiert in der Exec-Library eine Funktion, mit der sich bestimmte Offset-Einsprünge modifizieren sollen. Die Funktion sieht wie folgt aus:

SetFunktion

SetFunktion (Library, Offset, Einsprung)
A1 A0 D0

Offset: -420

Beschreibung: Die Funktion verändert den Einsprung der mit einem negativen Offset ausgesuchten Funktion so, daß nun der Einsprung der Funktion auf die neue Routine zeigt. Die Prüfsumme der Library wird neu errechnet.

Parameter:

Library ist ein Zeiger auf die zu ändernde Library.

Offset gibt den Offset der Funktion an, die geändert werden soll.

Einsprung ist ein Zeiger auf die eigene Routine.

2.3.5 Das Erstellen einer eigenen Library

Nachdem besprochen wurde, wie man Libraries nutzt, soll nun gezeigt werden, wie man sich seine eigene Library erstellt.

Das Erstellen einer eigenen Library ist dann sinnvoll, wenn mehrere parallel arbeitende Tasks gebraucht werden, die gemeinsam eine Anzahl von Funktionen benutzen, welche von einem der Tasks zur Verfügung gestellt werden. In diesem Fall empfiehlt es sich, eine Library der Funktionen zu erstellen, da so jeder Task bequem auf die Funktionen zugreifen kann.

Zum Erstellen einer Library stellt der Exec-Library einige Funktionen zur Verfügung, die vor der Erstellung der eigenen Library erklärt werden sollten.

InitStruct

InitStruct(initTabelle, Speicher, Größe)

A1

A2

D0

Offset: -78

Beschreibung: Die Funktion initialisiert eine Struktur ab dem angegebenen Speicherbereich nach einer angegebenen Tabelle.

Parameter:

initTabelle ist ein Zeiger auf die Tabelle, mit deren Hilfe eine Struktur erstellt wird.

Speicher ist ein Zeiger auf den bereits zugewiesenen Speicher.

Größe gibt die Größe der zu initialisierenden Struktur wieder. Der Speicher, in der die Struktur erstellt wird, braucht nicht gelöscht worden zu sein, denn dies wird von der InitStruck-Funktion übernommen.

Eine Tabelle, anhand der die Struktur erstellt wird, hat ein etwas verwirrendes Aussehen. Sie besteht aus einem Befehlsbyte, worauf Daten folgen, die je nach Aussehen des Befehlsbytes verschieden verarbeitet werden. Nach den Daten folgt wieder ein Befehlsbyte und Daten. Die Länge der Daten hängt wieder vom Befehlsbyte ab. Die Tabelle ist zu Ende, wenn das Befehlsbyte den Wert Null hat.

Das Befehlsbyte ist in High- und Low-Nibbel unterteilt. Die Bitkombinationen des High-Nibbels (die oberen vier Bits) geben den Befehl an und das Low-Nibbel die Anzahl der Befehlsausführungen.

Wenden wir uns zuerst dem High-Nibbel zu. Dieses ist wiederum in die zwei oberen und unteren Bits eingeteilt. Die oberen zwei Bits geben den eigentlichen Befehl und die unteren zwei des High-Nibbels die Datengröße an, mit der operiert werden soll. Als Datengröße stehen Langwort, Wort oder Byte zur Verfügung.

Mit den obersten zwei Bits lassen sich vier verschiedene Befehle kodieren:

Kombination 00

Die Kombination gibt an, daß die Daten, die nach dem Befehlsbyte beginnen, in die zu erstellende Struktur übertragen werden. Was für Daten verarbeitet werden (Langwort, Wort oder Byte), wird in den nachfolgenden zwei Bits entschieden.

Kombination 01

Die Kombination gibt an, daß das Datum der angegebenen Länge entsprechend oft in die zu erstellende Struktur kopiert wird.

Kombination 10

Die Kombination gibt an, daß das nach dem Befehlswort stehende Byte als Offset für die zu erstellende Struktur dient. Der Offset wird zur Startadresse der Struktur addiert und die nach diesem Byte stehenden Daten in die zu erstellende Struktur kopiert. Nach Beendigung des Befehls werden die folgenden Daten ab der Position in die Struktur geschrieben, an der der Befehl endete.

Kombination 11

Die Kombination gibt an, daß die drei Bytes nach dem Befehlsbyte als 24-Bit Offset verwendet werden sollen. Ansonsten ist dieser Befehl und der vorherige identisch.

Die nächsten zwei Bits des Befehlsbytes (die Bits 4 und 5) geben an, um welchen Datentyp es sich handelt.

Kombination: 00	Langwort (nur auf geraden Adressen)
Kombination: 01	Wort (nur auf geraden Adressen)
Kombination: 10	Byte
Kombination: 11	nicht verwenden, sonst Absturz

Das Low-Nibbel des Befehlsbytes gibt an, wie oft eine bestimmte Funktion ausgeführt werden soll; sie dient als Zähler. Da der Zähler bis -1 heruntergezählt wird, wird die Funktion somit einmal mehr ausgeführt als im Zähler angegeben.

Das Befehlsbyte darf immer nur auf einer geraden Adresse liegen.

Zur Verdeutlichung zwei Beispiele:

```
dc.b %00010010,$00
dc.w $FFFF,$FFFF,$1234
```

Das Befehlsbyte ist \$12 = %00010010 und legt somit fest, daß die folgenden drei Worte in die Struktur kopiert werden. Das Null-Byte nach dem Befehlsbyte ist notwendig, da die Worte nur auf einer geraden Adresse beginnen dürfen.

```
dc.b %10000001,$10
dc.l $12341234,$ffff1111
```

Das Befehlsbyte gibt an, daß zwei Langworte in die Struktur ab Position 16 kopiert werden.

Programmbeispiel:

In diesem Beispiel wird der Speicherplatz für eine Library-Struktur reserviert und diese teilweise initialisiert.

```

AllocMem = -198
FreeMem  = -210
MemType  = $10001
InitStruct = -78
StructSize = 34
Size     = $300

        move.l $4,a6
        move.l #MemType,d1
        move.l #Size,d0
        jsr AllocMem(a6)
        tst.l d0
        beq fehler
        lea.l Table,a1
        move.l d0,a2
        move.l StructSize,d0
        jsr InitStruct(a6)
fehler:  rts

name:    dc.b 'Test',0,0

Table:
dc.b %01000001,$00
dc.l 0                ;Node Succ und Pred
dc.b %00100001
dc.b $09,0,0          ;Type und Pri
dc.b %00011000,0
dc.l name              ;Name
dc.w 0,0,0,1,0         ;NegS,PosS,Version,Revision
dc.l name              ;IDString
dc.b %00000000         ;Endmarkierung

```

MakeLibrary

```
Library = MakeLibrary(Vektoren,Struktur,Init,Größe,SegListe)
D0                A0      A1      A2      D0      D1
```

Offset: -84

Beschreibung: Mit dieser Funktion ist es möglich, eine eigene Library zu erstellen. Der Speicherplatz, der Library-Struktur, wird von der Funktion selbstständig belegt sowie lib_NenSize und lib_PosSize korrekt initialisiert.

Parameter:

Vektoren ist ein Zeiger auf eine Tabelle der Vektoren für die Library. In der Tabelle stehen entweder direkt die Zeiger auf die verschiedenen Funktionen oder Offsets, die zu der Basisadresse der Library addiert werden, um den Einsprung für die Funktion zu erhalten. Wenn Sie in der Tabelle Offsets gespeichert haben, so muß die Tabelle mit \$FFFF (-1) begonnen werden. Die Endmarkierung für die Tabelle ist wieder -1 mit der gleichen Länge der Einträge in der Tabelle (Wortlänge für Offssettabelle, Langwort für Zeiger auf Funktionen).

Struktur ist ein Zeiger auf eine Initialisierungs-Tabelle, wie sie schon für die InitStrukt-Funktion beschrieben wurde. Die Library-Struktur wird mit dieser Tabelle erst am Ende der Funktion MakeLibrary() aufgebaut. Die Einträge lib_NegSize und lib_PosSize dürfen nicht mit Hilfe der Tabelle initialisiert werden, da sonst die von der Funktion errechneten und bereits eingefügten Werte wieder überschrieben werden. Ist der Parameter 'Struktur' in der MakeLibrary()-Funktion nicht gesetzt, wird keine Tabelle zur Initialisierung verwandt, wodurch das Erstellen der Struktur "von Hand" geschehen muß.

Init	ist ein Zeiger auf ein eigenes Programm, das am Ende der MakeLibrary()-Funktion, sofern der Zeiger gesetzt ist, aufgerufen wird. In einer eigenen Routine kann man beispielsweise die Library-Struktur initialisieren, wenn dies nicht schon mit Hilfe einer Initialisierungstabelle geschehen ist. Der Zeiger auf die zu erstellende Library-Struktur wird in D0 und der Zeiger auf die Segment-Liste in A0 übergeben. Sollte die Init-Routine D0 verändern, so wird diese Veränderung als Parameter beim Beenden der Funktion übergeben.
SegListe	ist ein Zeiger auf eine Segment-Liste (wird vom DOS benutzt), der bei Verwendung der Init-Routine in A0 übergeben wird.
Library	ist der zurückgegebene Zeiger auf die Library-Struktur, was nicht mit dem Anfang des für die Library belegten Speichers zu verwechseln ist.

Mit der zuvor besprochenen Funktion ist es möglich, eine eigene Library zu erstellen. Diese ist jedoch noch nicht in die LibListe der ExecBase-Struktur eingefügt. Außerdem ist die Prüfsumme der Library noch nicht berechnet. Für diese Aufgaben stellt die Exec-Library eine weitere Funktion zur Verfügung, die sich AddLibrary nennt.

AddLibrary

AddLibrary(Library)

A1

Offest: -396

Parameter:

Library	ist ein Zeiger auf die zuvor mit der MakeLibrary()-Funktion erstellten Library-Struktur.
----------------	--

Um das Erstellen einer eigenen Library und deren Benutzung zu verdeutlichen, folgt nun ein Beispielprogramm, das diese Aufgaben erfüllt.

```

MakeLib = -84
AddLib  = -396
OpenLib = -552
Blinken = -30
OpenCnt = 32
InitStruct = -78
StructSize = 34

        move.l $4,a6
        lea.l Vektoren,a0      ;Zeiger auf Vektoren
        move.l #0,a1          ;Struktur löschen
        lea.l init,a2         ;Init-Routine
        move.l #StructSize,d0 ;Strukturgröße
        clr.l d1              ;SegListe
        jsr MakeLib(a6)
        tst.l d0
        beq Fehler
        move.l d0,a1
        jsr AddLib(a6)
        lea.l name,a1         ;Name der Lib
        move.l #1,d0          ;Version
        jsr OpenLib(a6)
        tst.l d0
        beq Fehler
        move.l d0,LibBase
        move.l d0,a6
        move.l #$20000,d0      ;Wert für Blinken
        jsr blinken(a6)        ;Funktion aufrufen
Fehler:  rts

init:    move.l d0,a0
        move.b #9,8(a0) ;Typ eintragen
        lea.l name,a1
        move.l a1,10(a0) ;Name eintragen
        move.l a1,24(a0) ;IDString = Name
        move.w #1,20(a0) ;Version
        rts

open:    move.l a6,d0
        add.w #$01,OpenCnt(a6)
        rts

```



```
close:      sub.w #$01,OpenCnt(a6)
            clr.l d0
            rts
expunge:    clr.l d0
            rts
extfunc:    clr.l d0
            rts
blink:      move.w d0,$dff180
            sub.l #1,d0
            bne blink
            rts

LibBase:    dc.l 0
name:       dc.b 'test.library',0,0

Vektoren:
dc.l open, close,expunge,extfunc,blink,$fffffff
```

Das Programm erstellt eine Library des Namens "test.library". Es ist eine Funktion eingefügt, die mit Offset -30 zu erreichen ist. Die Funktion heißt Blinken und verändert kurzzeitig die Bildschirmfarbe. In D0 wird der Parameter für die Blinkdauer übergeben.

Es ist unbedingt erforderlich, daß die Library-Offsets -6 bis -24 mit den Einsprünge für die internen Funktionen belegt werden, wie auch schon zuvor beschrieben wurde.

Da die Strukturen Library, Resource und Device als standardmäßigen Kopf ihrer Struktur einer Library-Struktur beinhalten, ist es möglich mit der MakeLibrary()-Funktion auch eine Device oder Resource-Struktur zu erstellen.

2.3.6 Beschreibung der restlichen Library-Funktionen

RemLibrary

```
Error = RemLibrary(library)
      D0                A1
```

Offset: -402

Beschreibung: Die Funktion entfernt eine Library-Struktur aus der Library-Liste der ExecBase-Struktur. Nach dem Entfernen der Library ist es nicht möglich, sie mit OpenLibrary() zu öffnen.

Parameter:

Library ist ein Zeiger auf die Library-Struktur.

Error gibt an, ob in der Funktion ein Fehler aufgetreten ist. Ist das der Fall, wird die Fehlermeldung in D0 übergeben, ansonsten wird Error auf Null gesetzt.

OldOpenLibrary

```
Library = OldOpenLibrary(libName)
      D0                A1
```

Offset: -408

Beschreibung: Die Funktion ist ein Überbleibsel aus der Kickstartversion 1.0. Sie dient ebenfalls zum Öffnen einer Library, prüft jedoch nicht die Versionsnummer der zu öffnenden Library. Die Funktion wurde nur deshalb in der ExecLibrary belassen, damit Programme, die mit der Kickstartversion 1.0 arbeiten, auch mit der Version 1.2 laufen.

2.4 Multitasking

Das Multitasking ist eine der hervorstechendsten Fähigkeiten von Exec. Man versteht darunter die Fähigkeit des Betriebssystems, mehrere Programme gleichzeitig ablaufen zu lassen. Jedes dieser Programme stellt einen sogenannten Task dar. Da der Amiga nur über einen Prozessor verfügt, kann in Wirklichkeit natürlich immer nur ein Task tatsächlich arbeiten. Damit trotzdem der Eindruck eines gleichzeitigen Ablaufs mehrerer Tasks entsteht, wird der 68000er zwischen den Tasks aufgeteilt. Dies kann nur in einem zeitlichen Rahmen geschehen, d.h. jeder Task bekommt den Prozessor für eine bestimmte Zeitspanne. Der Prozessor wird im sogenannten Zeitmultiplexverfahren den Tasks zugeteilt.

Nicht alle Tasks, die sich im Computerspeicher befinden, müssen gleichzeitig ablaufen. Viele von ihnen werden nur im Bedarfsfall aktiviert. Sie können sonst auf den Druck einer bestimmten Taste, auf eine Mausposition usw. warten.

Aus diesen Gründen werden die verschiedenen Tasks in folgende Kategorien (engl. Task states) eingeteilt:

- | | |
|---------|---|
| running | Dies ist der Task, dem gerade der Prozessor zugeteilt ist. Es befindet sich immer nur ein einziger Task im running state. |
| ready | Alle Tasks, die bereit zum Ablauf sind, denen der Prozessor zur Zeit aber nicht zugeteilt ist, befinden sich im ready state. |
| waiting | Tasks im waiting state sind all diejenigen, die zur Zeit nicht abgearbeitet werden sollen, da sie auf ein bestimmtes Ereignis warten. |

Zusätzlich kann sich ein Task vorübergehend auch in einem der folgenden Zustände befinden:

- added** Ein solcher Task wurde gerade zum System hinzugefügt und ist noch nicht in einem der drei obigen Zustände.
- removed** Dieser Task ist gerade beendet worden. Er befindet sich nicht mehr in einem der drei oberen Zustände und wird aus dem System entfernt.
- exception** Eine Task Exception ist ein besonderer Zustand, in den der Task durch ein bestimmtes Ereignis versetzt werden kann. Nach Bearbeitung dieser Exception kehrt er wieder zu einem der drei oberen Zustände zurück.

Ein Task besteht im wesentlichen aus zwei Elementen: Dem eigentlichen Programm und der Task-Struktur. Diese enthält alle Informationen über den Task, die Exec benötigt. Am besten versteht man die vielfältigen Möglichkeiten eines Tasks, wenn man die Task-Struktur genau betrachtet:

2.4.1 Die Task-Struktur

Die Task-Struktur, wie sie in dem Include-File "exec/tasks.h" eines C-Compilers definiert ist, sieht folgendermaßen aus (Die Zahlen in Klammern bezeichnen die Entfernung eines Elements von der Basisadresse der Struktur):

```
extern struct Task {
    struct Node  tc_Node;
    UBYTE  tc_Flags;          /* (14)
    UBYTE  tc_State;          /* (15) task state
    BYTE   tc_IDNestCnt;      /* (16) Zähler für Disable() */
    BYTE   tc_TDNestCnt;      /* (17) Zähler für Forbid() */
    ULONG  tc_SigAlloc;       /* (18) besetzte Signalbits */
    ULONG  tc_SigWait;        /* (22) auf diese wird gewartet */
    ULONG  tc_SigRecvd;       /* (26) empfangene Signale */
    ULONG  tc_SigExcept;      /* (30) Exceptions auslösende S.*/
    UWORD  tc_TrapAlloc;      /* (34) besetzte Trap-Befehle */
}
```

```
UWORD tc_TrapAble;      /* (36) erlaubte Trap-befehle */
APTR tc_ExceptData;     /* (38) Daten für Exceptions */
APTR tc_ExceptCode;     /* (42) Code für Exceptions */
APTR tc_TrapData;       /* (46) Daten für Traphandler */
APTR tc_TrapCode;       /* (50) Code für Traphandler */
APTR tc_SPReg;          /* (54) Zwischenspeicher für SP */
APTR tc_SPLower;        /* (58) untere Stack-Grenze */
APTR tc_SPUpper;        /* (62) obere Stack-Grenze +2 */
VOID (*tc_Switch)();    /* (66) Task verliert CPU */
VOID (*tc_Launch)();    /* (70) Task erhält CPU */
struct List tc_MemEntry; /* (74) belegter Speicher */
APTR tc_UserData;       /* (88) Zeiger auf Task-Daten */
};
```

Wie man sieht, besteht der Kopf einer Task-Struktur aus einer Node-Struktur, wie wir sie im vorangegangenen Kapitel kennengelernt haben. Der Grund dafür ist, daß Exec die Tasks in zwei verschiedenen Listen verwaltet. Jeweils eine für die ready-Tasks und eine für die waiting-Tasks. Man kann auf eine Task-Liste daher auch prinzipiell Funktionen wie Insert(), Remove() oder FindName() anwenden.

Allerdings existieren eigene Funktionen zur Verwaltung der Task-Listen, die speziell an die Tasks angepaßt sind.

Zu jeder Liste gehört bekanntlich auch ein Listen-Kopf. Im Falle der Task-Listen befinden sich diese in der ExecBase-Struktur. Auf diese Struktur soll an dieser Stelle nicht näher eingegangen werden. Sie wird später noch genau beschrieben. Folgende Zeilen zeigen, wie man auf die Task-Listen zugreifen kann:

```
#include <exec/execbase.h>

extern ExecBase *SysBase;

main()
{
    struct Task *waiting, *ready, *running;

    waiting=(struct Task *)SysBase->TaskWait.lh_Head;
    ready=(struct Task *)SysBase->TaskReady.lh_Head;
    running=(struct Task *)SysBase->ThisTask;

    usw.
}
```

Dieses Programm übergibt in waiting und ready je einen Zeiger auf die erste Task-Struktur der jeweiligen Liste. Bei running handelt es sich um einen Zeiger auf den running Task. TaskWait und TaskReady sind also Listen-Köpfe, wobei es sich bei ThisTask nur um einen Zeiger handelt. Da es immer nur einen einzigen running Task geben kann, existiert hier natürlich keine Liste.

Task-Switching

An dieser Stelle sind jetzt erst einmal ein paar Worte zu dem zentralen Vorgang nötig, der dafür verantwortlich ist, daß mehrere Tasks denselben Prozessor benutzen können: dem Task-Switching. Wie der Name schon sagt, handelt es sich dabei um das Umschalten der verschiedenen Tasks. Befindet sich ein Task einmal in einer der beiden Task-Listen, wird er automatisch in diesen Vorgang mit aufgenommen. Frage Nr. 1 beim Task-Switching ist:

Wie werden die Tasks umgeschaltet?

Ein Task weiß nicht, wann er den Prozessor erhält und wieder abgeben muß. Prüft er das ThisTask-Feld in ExecBase, findet er dort immer den Zeiger auf seinen Task, da er dieses Feld ja nur abfragen kann, wenn ihm gerade die CPU zugeteilt ist. In Wirklichkeit kann er aber zu jedem beliebigen Zeitpunkt unterbro-

chen werden. Ausgelöst wird der ganze Vorgang von einem Interrupt, der auftritt, wenn der Task den ihm zustehenden Anteil an Rechenzeit verbraucht hat oder wenn ein wichtiger Task sofort abgearbeitet werden muß. Die Switch-Routine des Betriebssystems erledigt das eigentliche Umschalten. Im Gegensatz zum Task, der immer im User-Mode des 68000er läuft, wird die Switch-Routine im Supervisor-Modus abgearbeitet. Zuerst werden die Prozessorregister D0-D7 und A0-A6 auf dem Task-Stack gesichert. Der User-Stackpointer kommt dann in das `tc_SPReg`-Feld der Task-Struktur, und zuletzt kommen noch Statusregister und Programmzähler auf den User-Stack. Sie wurden vom 68000 durch den Interrupt automatisch auf den Supervisor-Stack gelegt. Dann wird der Task in die Ready-Liste übernommen.

Der neue Task stammt entweder aus der Ready- oder Waiting-Liste. Er wird jetzt dort entfernt und stattdessen in das `ThisTask`-Feld eingetragen. Danach werden erst sein Stackpointer aus seinem `tc_SPReg`-Feld und anschließend seine Register von seinem Stack geholt. Exec verläßt die Switch-Routine am Ende mit einem RTE. Damit wird automatisch der neue Task bearbeitet.

Dies war selbstverständlich nur eine grobe Zusammenfassung des Ablaufs der Switch-Routine. Tatsächlich werden noch einige Daten ausgetauscht und Sonderfälle überprüft. Befindet sich der Task im Exception State, kann es vorkommen, daß die Switch-Routine vor bzw. nach dem Umschalten noch die Routinen aufruft, deren Adressen in den `tc_Launch` und `tc_Switch` Einträgen der Task-Struktur gespeichert sind.

Aber wie gesagt, der gesamte Umschaltvorgang läuft automatisch ab. Sie brauchen sich darum nicht zu kümmern.

Die zweite Frage beim Task-Switching ist: Wann werden die Tasks umgeschaltet?

Formuliert man die Frage anders, lautet sie: Welchen Anteil der Rechenzeit bekommt ein bestimmter Task? Das Verteilen der Rechenzeit heißt Task-Scheduling. Als Grundlage dafür verwen-

det Exec das `ln_Pri` Feld der Listnode-Struktur am Anfang der Task-Struktur. Allgemein gilt, je höher die Priorität eines Tasks, desto mehr Rechenzeit bekommt er, d.h. desto schneller wird er abgearbeitet. Exec beginnt mit dem Task mit der höchsten Priorität. Er bekommt für eine bestimmte Zeitspanne den Prozessor. Danach wird die Rechenzeit, die er verbraucht hat, von seiner relativen Priorität im Vergleich zu den ready-Tasks abgezogen. Damit ist er jetzt nicht mehr der Task mit der höchsten Priorität, und so kommt der nächste Task an den Prozessor. Dieses Verfahren stellt sicher, daß auch Tasks mit einer niedrigen Priorität ab und zu einmal den Prozessor bekommen und daß zeitkritische Tasks, die den Prozessor zwar sofort, aber nur für eine kurze Zeitspanne, benötigen, den anderen vorgezogen werden.

Die Priorität kann wie üblich Werte zwischen -128 und 127 annehmen. Ein normaler Task sollte die Priorität 0 erhalten. Dies ist auch die Priorität eines CLI-Tasks. Die übrigen Tasks des Betriebssystems bewegen sich zwischen -20 und 20, es ist also nicht nötig, extreme Werte in das `tc_Node.ln_Pri` Feld einer Task-Struktur zu schreiben.

Ein weiteres Feld der Task-Struktur ist `tc_Node.ln_Name`. Es enthält einen Zeiger auf den Namen des Tasks (siehe Kapitel über Listen). Damit wird das Auffinden eines bestimmten Tasks erleichtert.

Das `tc_State` Feld enthält den Task State. Folgende Werte sind den verschiedenen States zugeordnet:

ungültig	= 0
added	= 1
running	= 2
ready	= 3
waiting	= 4
exception	= 5
removed	= 6

Der Task-Stack

Neben der Task-Struktur benötigt der Task auch einen Stack. Dies ist, wie schon erwähnt, ein User-Stack. `Tc_SPLower` enthält die untere, `tc_SPUpper` die obere Grenze des Stacks (zur Erinnerung: ein Stack wächst immer von oben nach unten, d.h. von den hohen Adressen zu den niedrigen).

Die Adresse in `tc_SPReg` wird als Speicher für den Stackzeiger verwendet. Normalerweise setzt man `tc_SPReg = tc_SPUpper`. Man kann `tc_SPReg` aber auch auf eine beliebige Adresse zwischen `tc_SPUpper` und `tc_SPLower` setzen. Dann kann man den Bereich zwischen `tc_SPReg` und `tc_SPUpper` als Speicher für globale Variablen oder ähnliches benutzen.

Noch ein Wort zu `tc_SPUpper`:

`Tc_SPUpper` zeigt auf die obere Grenze des Stacks, damit ist die erste Adresse nach dem Stackbereich gemeint. Das Wort an der letzten Position im Stack hat daher die Adresse `tc_SPUpper` minus 2.

Da Exec beim Task-Switching die Register auf dem Task-Stack zwischenspeichert, muß dieser eine Mindestgröße von 70 Bytes haben. Dann darf der Task aber weder Variablen noch Rücksprungadressen auf den Stack legen. Da aber besonders C-Programme reichlich vom Stack gebrauch machen, sollte man ihn mit mindestens einem Kilobyte bemessen.

Da ein Task aus verschiedenen Elementen, Task-Struktur, Stack, Programm usw. besteht, muß man auch mehr oder weniger Speicher für ihn reservieren. Es besteht daher die Möglichkeit, eine Liste, die all den belegten Speicher des Tasks enthält, in der Task-Struktur unterzubringen. Das `tc_MemEntry` Feld enthält den zugehörigen Listenkopf. Wir wollen in diesem Kapitel auf diese Möglichkeit nicht weiter eingehen, da der Aufbau einer solchen Speicher- oder MemList erst später erklärt wird.

Erlauben und Verboten des Task-Switching

Es kann manchmal störend sein, daß ein Task zu jedem beliebigen Zeitpunkt den Prozessor verlieren kann. Stellen Sie sich vor, Sie wollen die Liste der Waiting Tasks auf dem Bildschirm ausgeben. Aber während Ihr Programm Eintrag für Eintrag liest, wechselt ein Ready Task zum Waiting State oder umgekehrt. Dann sind die gelesenen Werte falsch.

Daher gilt grundsätzlich: Immer wenn ein Task auf Datenstrukturen zugreift, die entweder dem gesamten System oder bestimmten anderen Tasks offenstehen, muß vorher das Task-Switching abgeschaltet werden, damit die Daten während des Zugriffs nicht von einem anderen Task oder dem Betriebssystem verändert werden können.

Forbid() und Permit()

Diese beiden Routinen stellen die erste Stufe der oben erwähnten Abschaltung dar. Forbid() sperrt das Task-Switching, Permit() erlaubt es wieder. Beide Routinen werden ohne Parameter aufgerufen und geben auch keine Werte zurück (C-Typ: void).

Disable() und Enable()

Oft reicht es nicht aus, nur das Task-Switching abzuschalten. Denn viele Datenstrukturen des Systems werden von Exec innerhalb von Interrupts verändert. Daher kann man sämtliche Interrupts mit Disable() verbieten und mit Enable() wieder erlauben. Aber Vorsicht! Ein Verbot des Task-Switchings, auch über längere Zeit hinweg, schadet nicht. Anders ist es bei den Interrupts. Ihr regelmäßiges Auftreten ist für Exec lebenswichtig. Sperrt man die Interrupts für längere Zeit, kann dies zu einem Systemzusammenbruch führen, wenn man später wieder zum Multitasking zurückkehren will. Während eines Disable() ist der Task völlig ungestört, da durch das Abschalten der Interrupts auch kein Task-Switching mehr stattfindet.

Es ist auch möglich, mehrere `Forbid()`- oder `Disable()`-Aufrufe ineinander zu verschachteln. Es existieren zwei Zähler: `TDNestCnt` und `IDNestCnt` (Task Disable Nesting Counter und Interrupt Disable Nesting Counter). Mit jedem Aufruf von `Forbid()` wird `TD_NestCnt` um 1 erhöht, bei `Disable` um 1 erniedrigt. Task-Switching ist nur möglich, wenn `TDNestCnt < 0` ist. Dies bedeutet, daß die Anzahl der `Permit()` Aufrufe gleich derjenigen der `Forbid()` sein muß, bevor das Task-Switching wieder erlaubt wird.

Gleiches gilt für `Enable()` und `Disable()`.

Folgendes Programm zeigt die Verwendung der `Enable()`- und `Disable()`-Funktionen. Es liest die Zeiger auf sämtliche Task-Strukturen der Ready und Waiting Tasks in ein Feld, während die Interrupts mittels `Disable()` abgeschaltet sind. Danach werden sie wieder erlaubt (`Enable()`), und mittels der im Feld gespeicherten Zeiger werden die wichtigen Felder der Task-Strukturen, wie Name, Stack und Priorität, auf dem Bildschirm ausgegeben.

Das Programm zeigt auch sehr schön, welche Tasks beim Amiga immer vorhanden sind. Experimentieren sie mit CLI-Befehlen wie `NEWCLI`, `RUN` oder `SETTASKPRI`.

Auch der Running Task wird mit ausgegeben, logischerweise ist dies immer der Task, von dem unser Programm aufgerufen wurde.

```
#include <exec/execbase.h>

struct ExecBase *SysBase;

main()
{
    register struct Task *a_task;
    APTR run, tnodes[50], wtask, ltask;
    void o1(), o2();
    register APTR Anode;

    Disable();
```

```

Anode = tnodes;
run = (APTR)SysBase->ThisTask;

for(a_task=(struct Task *)SysBase->TaskReady.lh_Head;
a_task->tc_Node.ln_Succ;
*Anode=(APTR)a_task,Anode++,
a_task=a_task->tc_Node.ln_Succ;
wtask=Anode;

for(a_task=(struct Task *)SysBase->TaskWait.lh_Head;
a_task->tc_Node.ln_Succ;
*Anode=(APTR)a_task,Anode++,
a_task=a_task->tc_Node.ln_Succ;
ltask=Anode;

Enable();

printf("\nTask in the running state:\n");o1();o2(run);

printf("\nTask(s) in the ready state:\n");o1();
for(Anode=tnodes;Anode!=wtask;o2(*Anode),Anode++);

printf("\nTask(s) in the waiting state:\n");o1();
for(;Anode!=ltask;o2(*Anode),Anode++);

}

void o1()
{
printf
("Stackaddress Stacksize Priority Signals Name\n");
printf
("-----\n");
}

void o2(at)
register struct Task *at;
{
printf("%10lx%10lx%8ld%11lx %s\n",at->tc_SPLower,
(ULONG)at->tc_SPUpper - (ULONG)at->tc_SPLower - 2L,
(LONG)at->tc_Node.ln_Pri,
at->tc_SigWait,at->tc_Node.ln_Name);
}

```

Da das Programm lediglich die Zeiger auf die Task-Strukturen sichert, nicht aber den Inhalt dieser Strukturen, besteht immer

noch eine potentielle Fehlermöglichkeit, nämlich wenn ein Task während der Ausgabe der Werte gelöscht wird. Da dies aber so gut wie nie vorkommt und es ein riesiger Mehraufwand wäre, alle Task-Strukturen zwischenspeichern, wurde in dem Programm auch darauf verzichtet. Durch Abändern der o1()- und o2()-Routinen kann man auch andere Felder der Task-Strukturen ausgeben. Experimentieren Sie ruhig etwas.

Erzeugen neuer Tasks

Nachdem soviel über Tasks und Task-Strukturen geredet wurde, soll jetzt ein neuer Task kreiert werden. Was wird dafür benötigt?

Zuerst eine Task-Struktur, dann ein Stack, der Taskname, denn die Task-Struktur enthält ja nur einen Zeiger auf den eigentlichen Namen, und zuletzt noch ein Programm, den eigentlichen Task.

Hier tauchen jetzt einige Probleme auf. Es wird eigener Speicher für den Task benötigt, denn er soll ja auch nach Beendigung des Programms, das ihn erzeugt hat, im Speicher bleiben.

Um das Programm nicht komplizierter als nötig zu machen, werden Task-Struktur, Taskname und Stack zu dem neuen Strukturtyp "alltask" zusammengefaßt, für den dann ein gemeinsamer Speicherblock belegt wird.

Der eigentliche Task ist als normale C-Funktion mit dem Namen "code" geschrieben. Er macht nichts weiter, als einen Zähler hinaufzuzählen, bis dieser den Wert \$FFFFFF erreicht hat. Dafür benötigt er allerdings einige Minuten. Danach endet der Task. Seine Anwesenheit bemerkt man am Langsamerwerden des Amiga, da unser Task ja auch seinen Anteil an Rechenzeit bekommt. Oder man verwendet das vorangegangene Beispielprogramm, das alle Tasks auflistet. Unser Task erscheint als Ready Task mit dem Namen "Beispiel-Task".

Um die code-Funktion an ihre engültige Speicherposition zu kopieren, wird ihr Name als Zeiger auf ihre Adresse verwendet.

Die Funktion "end" dient nur dazu, die Endadresse der code-Funktion zu erhalten. Da es in C keine Möglichkeit gibt, den Speicherbedarf einer Funktion zu ermitteln, wird er durch Bildung der Differenz aus Anfangs- und Endadresse errechnet.

Auf eine Freigabe des belegten Speichers wurde verzichtet, um das Programm nicht noch komplizierter zu machen. Es handelt sich dabei auch nur um knapp ein Kilobyte.

```

/**** Erzeugen eines Tasks ****/

#include <exec/types.h>
#include <exec/Tasks.h>
#include <exec/memory.h>

#define STACK_SIZE 500 /* Größe des Stacks */

main()
{
    void code(),end();
    APTR mycode, AllocMem();
    static char Taskname[] = "Beispiel-Task";
    register APTR c1,c2;

    struct alltask {
        struct Task tc;
        char    Name[sizeof(Taskname)], Stack[STACK_SIZE];
    } *mytask;

    mytask = AllocMem((ULONG)sizeof(*mytask),
        MEMF_PUBLIC|MEMF_CLEAR);
    if(mytask==0)
    {printf("Kein Speicher fuer die AllTask-Struktur!\n");
        return(0);
    }

    mycode = AllocMem((ULONG)end-(ULONG)code, MEMF_PUBLIC);
    if(mycode==0)
    {FreeMem(mytask,(ULONG)sizeof(*mytask));
        printf("Kein Speicher fuer den Task-Code!\n");
        return(0);
    }

    strcpy(mytask->Name,Taskname);

```

```
mytask->tc.tc_SPLower=mytask->Stack;
mytask->tc.tc_SPUpper=mytask->Stack+STACK_SIZE;
mytask->tc.tc_SPReg=mytask->tc.tc_SPUpper;

mytask->tc.tc_Node.ln_Type=NT_TASK;
mytask->tc.tc_Node.ln_Name=mytask->Name;

for(c1=code,c2=mycode;c1<=end;*c2++=*c1++);

AddTask(mytask,mycode,0L);
}

/**** Die Funktion "code" ist der eigentliche Task ****/

void code()
{
    ULONG count;

    for(count=0;count<0xffffffff;count++);
}

void end(){}

```

Wie man sieht, müssen nur wenige Felder der Task-Struktur initialisiert werden:

- tc_SPLower mit der unteren Stack-Grenze
- tc_SPUpper und tc_SPReg mit der oberen Stack-Grenze
- tc_Node.ln_Type mit dem Listentyp NT_TASK

Auf den Namen kann man auch verzichten.

Eine wesentliche Funktion wurde in diesem Programm zum erstenmal verwendet:

AddTask(task, initialPC, finalPC)

Diese Funkiton fügt einen neuen Task zum System hinzu. Normalerweise wird der neue Task sofort zur Ready-Liste hinzugefügt.

Die AddTask-Funktion benötigt folgende Parameter:

task	Dies ist ein Zeiger auf eine Task-Struktur, in der mindestens die vier oben erwähnten Felder initialisiert sein sollten.
initialPC	Dies ist die Adresse, bei der die Abarbeitung des Task-Programms beginnen soll. In unserem Beispiel die Anfangsadresse der code-Funktion an ihrer endgültigen Adresse, die in mycode gespeichert ist.
finalPC	finalPC enthält die Adresse, die angesprungen wird, wenn der Task einen RTS-Befehl ausführt. Man kann hier die Adresse einer Routine einsetzen, die den belegten Speicher zurückgibt, geöffnete Files schließt usw. Gibt man einfach 0 als finalPC an (wie in unserem Beispiel), verwendet Exec seine Standard-finalPC-Routine. Diese gibt den Speicher frei, auf den das tc_MemEntry-Feld der Task-Struktur zeigt. Danach entfernt sie den Task aus den Systemlisten.

Beenden eines Tasks

Die Beschreibung von finalPC bringt uns auf das nächste Thema. Wie wird ein Task beendet? Folgende Möglichkeiten existieren:

1. Der Task erreicht einen RTS-Befehl, der nicht den Rücksprung eines taskeigenen JSR oder BSR darstellt. Dann läuft der unter finalPC geschilderte Vorgang ab.
2. Es tritt eine 68000er Exception auf, die nicht vom Task gehandhabt wird, z.B. ein Bus- oder Adresserror, Division by Zero usw. Exec erzeugt dann seine "Software Error - Task Held" Meldung oder eine Guru-Meditation. Am Ende dieses Kapitels wird

übrigens gezeigt, wie man solche Fehler auffangen kann.

3. Ein Aufruf der RemTask-Funktion, die den Task aus dem System entfernt.

2.4.1.1 Task-Funktionen

Für das Erzeugen und Löschen von Tasks und die Verwaltung der Task-Listen gibt es einige Funktionen im Exec:

AddTask

AddTask(task, initialPC, finalPC)
 A0 A1 A2

Offset: -282

Beschreibung: AddTask fügt einen neuen Task zum System hinzu.

Parameter:

task ist ein Zeiger auf eine Task-Struktur. Die Felder Tc_SPUpper, tc_SPLower, tc_SPReg und tc_Node.In_Type sollten vorher korrekt initialisiert werden.

initialPC ist die Adresse, bei der mit der Abarbeitung des Task-Programms begonnen wird.

finalPC ist die Rücksprungadresse, die vor Beginn des Tasks auf den Stack gelegt wird. Führt der Task ein überzähliges RTS aus, wird diese Adresse angesprungen. Ist finalPC auf 0, setzt Exec die Adresse seiner Standard-finalPC-Routine ein.

FindTask

```
Task = FindTask(Name)
```

```
DO          A1
```

Offset: -294

Beschreibung: FindTask durchsucht die Task-Listen nach einem Task mit dem angegebenen Namen und gibt, wenn es einen solchen findet, einen Zeiger auf dessen Task-Struktur zurück. Gibt man 0 als Name an, bekommt man in task einen Zeiger auf die Task-Struktur des eigenen Tasks.

Parameter:

Name ist ein Zeiger auf den Namen des zu suchenden Tasks oder 0.

Ergebnis:

Task ist ein Zeiger auf die Task-Struktur des gesuchten Tasks.

RemTask

```
RemTask(Task)
```

```
A1
```

Offset: -288

Beschreibung: RemTask entfernt einen Task aus dem System. Zeigt das tc_MemEntry Feld auf eine MemEntry-Liste, wird diese freigegeben. Alle sonst noch vom Task belegten Systemressourcen müssen vorher dem System zurückgegeben werden.

Parameter:

Task Zeiger auf die Task-Struktur des zu entfernennden Tasks. Ist Task = 0 wird der eigene Task gelöscht.

SetTaskPri

altePriorität = SetTaskPri(Task, neuePriorität)
D0 A1 D0

Offset: -300

Beschreibung: SetTaskPri gibt die alte Priorität eines Tasks zurück und setzt sie dann auf den neuen Wert. Dabei wird noch ein sogenanntes Rescheduling ausgeführt, dies bedeutet, daß die Rechenzeiten der einzelnen Tasks auf Grund der geänderten Priorität neu verteilt werden. Setzt man mit SetTaskPri einen Task auf eine hohe Priorität, bekommt er meistens sofort den Prozessor.

Parameter:

task Zeiger auf die Task-Struktur des Tasks
neuePriorität Neue Priorität des Tasks (in den unteren 8 Bits von D0)

Ergebnis:

altePriorität Alte Priorität des Tasks (in den unteren 8 Bits von D0)

Forbid

Forbid()

Offset: -132

Beschreibung: Verboten des Task-Switchings und erhöhen des TDNestCnt.

Permit

Permit()

Offset: -138*Beschreibung:* Herunterzählen des TDNestCnt und Task-Switching wieder erlauben, wenn TDNestCnt < 0 ist.**Disable**

Disable()

Offset: -120*Beschreibung:* Verboten aller Interrupts und erhöhen des IDNestCnt.**Enable**

Enable()

Offset: -126*Beschreibung:* Herunterzählen des IDNestCnt und Interrupts wieder erlauben, wenn IDNestCnt < 0 ist.**2.4.2 Kommunikation zwischen Tasks**

Nicht jeder Task ist dafür geschaffen, sein Dasein als Einzelgänger zu fristen. Er will Daten mit anderen Tasks austauschen. Meistens handelt es sich dabei um Ein-/Ausgabe-Vorgänge, denn die Routinen, die die verschiedenen Ein- und Ausgabegeräte, wie Tastatur, Bildschirm oder Diskette, steuern, sind als eigene Tasks ausgelegt.

Es wurde ja schon erwähnt, daß manche Tasks nur auf Signale anderer Tasks warten, um in Aktion zu treten. Und diese Signale sind auch das Thema des nächsten Abschnitts.

2.4.2.1 Die Task-Signale

Jeder Task besitzt 32 Signalbits, damit ist er in der Lage, verschiedene Ereignisse zu unterscheiden. Jeder Task kann diese Signale beliebig verwenden. Ein bestimmtes Signal mag für zwei verschiedene Tasks eine völlig andere Bedeutung haben. Allerdings benutzen einige Systemfunktionen, z.B. aus Intuition, bestimmte Signale für ihre Mitteilungen. Will ein Task eines seiner Signale benutzen, muß er es daher zuerst besetzen. Dies geschieht mit der AllocSignal-Funktion. Normalerweise sind die unteren 16 Bits für Systemfunktionen reserviert, damit bleiben noch 16 weitere für die freie Benutzung übrig.

Man kann mit AllocSignal() ein bestimmtes Signal besetzen, indem man die Nummer des gewünschten Signals als Argument übergibt, oder man läßt AllocSignal selbst das nächste freie Signal suchen, indem man -1 angibt.

Als Ergebnis erhält man immer die Nummer des gewünschten Signals, falls es vorher noch nicht besetzt war, oder aber -1 als Fehlermeldung. AllocSignal(-1) gibt nur -1 zurück, wenn überhaupt kein Signal mehr frei war.

Folgendes kleines C-Programm besetzt das nächstbeste freie Signal mit der AllocSignal-Funktion:

```
Signal=AllocSignal(-1L);
Signal<0 ?
printf("Keine freien Signale mehr vorhanden!"):
printf("Das Signal Nummer %ld wurde besetzt!",(long)Signal);
```

Grundsätzlich gibt es zwei Möglichkeiten, ein bestimmtes Signal anzugeben:

Entweder man gibt seine Nummer an. Dies ist eine Zahl zwischen 0 und 31, die der Bitnummer des entsprechenden Signals entspricht.

Die zweite Möglichkeit ist die, das gesamte Signallangwort anzugeben. In dieser sogenannte Signalmaske spiegelt der Zustand eines Bits dann den des entsprechenden Signals wieder. Der

Vorteil bei der Angabe der gewünschten Signale in Form einer Signalmaske ist, daß man mehrere Signale auf einmal wählen kann. AllocSignal gibt die Signalnummer zurück. Um von ihr zu der Signalmaske zu kommen, muß man das Bit an der durch die Signalnummer bezeichneten Bitposition setzen. Dies kann in C folgendermaßen geschehen:

```
Signalmaske=1<<Signalnummer;
```

In Maschinensprache genügt:

```
MOVE.W Signalnummer,D0  
MOVE.L Signalmaske,D1  
BSET D0,D1
```

wobei Signalmaske und Signalnummer jeweils die Adresse des jeweiligen Wertes darstellen (nicht den Wert selbst).

Die besetzten Signale eines Tasks werden im tc_SigAlloc Feld der Task-Struktur in Form einer Signalmaske gespeichert.

Warten auf Signale

Die wesentliche Funktion eines Signals ist, daß ein Task darauf warten kann. Dieser Ausspruch hört sich seltsam an, ist aber korrekt. Nehmen wir an, ein Task warte auf eine bestimmte Taste. Prinzipiell könnte er dies in Form einer Schleife tun. Dann würde er aber unnötig Rechenzeit verbrauchen, ohne daß tatsächlich etwas getan würde. Um dies zu verhindern, kann ein Task auf ein Ereignis, wie z.B. eine Taste, mit Hilfe der Task-Signale warten. Jeder Task kann auf seine Signale warten. Während er wartet, kommt er in die schon erwähnte Waiting-Liste und benötigt so keine Rechenzeit.

Um einen Task auf eines seiner Signale warten zu lassen, gibt es die Wait-Funktion. Sie benötigt nur ein einziges Parameter, nämlich eine Signalmaske, die alle Signale enthält, auf die gewartet werden soll. Es ist somit möglich, auf mehrere Signale gleichzeitig zu warten. Sobald eines der angegebenen Signale von einem anderen Task gesetzt wird, kehrt die Wait-Funktion

zurück und übergibt als Ergebnis eine Signalmaske, die das (oder auch die) aufgetretenen Signal(e) enthält. Mittels AND-Verknüpfungen zwischen den gewünschten Signalen und der von Wait() zurückgegebenen Signalmaske kann man feststellen, welches der Signale, auf die man gewartet hat, tatsächlich aufgetreten ist. Das folgende hypothetische Beispielprogramm in C soll dies demonstrieren:

```
unsigned long Signale;  
Signale=Wait(Taste|Mausknopf|Menue);  
if(Signale & Taste){ /* Taste gedrückt */ }  
if(Signale & Mausknopf){ /* Mausknopf gedrückt */ }  
if(Signale & Menue){ /* Menuepunkt aktiviert */ }
```

Ist eines der gewünschten Signale schon vor dem Aufruf gesetzt, kehrt Wait() sofort ins Programm zurück.

Die Signale, auf die ein Task wartet, werden in seinem tc_SigWait-Feld gespeichert, diejenigen, die er empfangen hat, in tc_SigRecvd.

Hat man vor dem Aufruf einer Wait-Funktion das Task-Switching oder die Interrupts abgeschaltet, werden sie durch die Wait-Funktion wieder erlaubt. Erst wenn Wait() wieder ins Programm zurückkehrt, wird der verbotene Zustand erneut hergestellt.

Bewerkstelligt wird dies, indem bei einem Wait()-Aufruf die Inhalte der beiden Zähler TDNestCnt und IDNestCnt in die zugehörigen Felder der Task-Struktur übertragen werden: tc_TDNestCnt und tc_IDNestCnt.

Es gibt fünf wesentliche Routinen im Zusammenhang mit den Task-Signalen. Wobei SetSignals() vom normalen Anwender nicht benötigt wird.

Die Signal-Funktionen

AllocSignal

```
Signalnummer= AllocSignal(Signalnummer)
                                D0          D0
```

Offset: -330

Beschreibung: Mittels der AllocSignal-Funktion kann man eines der Task-Signale besetzen. Gibt man statt der Nummer des zu besetzenden Signals -1 an, sucht AllocSignal das nächste freie Signal und belegt dieses. Ist das gewünschte Signal schon besetzt, gibt AllocSignal -1 zurück.

AllocSignal läßt sich nur auf die Signale des eigenen Tasks anwenden. Außerdem sollte AllocSignal nicht innerhalb einer Exception aufgerufen werden.

Parameter:

Signalnummer Die Nummer des zu besetzenden Signals (0 - 31) oder -1 für das nächste freie Signal.

Ergebnis:

Signalnummer Die Nummer des besetzten Signal oder -1, wenn das gewünschte Signal (oder alle Signale bei AllocSignal(-1)) schon belegt war.

FreeSignal

```
FreeSignal(Signalnummer)
                                D0
```

Offset: -336

Beschreibung: FreeSignal() ist die Umkehrfunktion zu AllocSignal. Das Signal mit der angegebenen Signalnummer wird freigegeben. Wie auch

AllocSignal() sollte FreeSignal() nicht aus einer Exception heraus aufgerufen werden.

Parameter:

Signalnummer Die Nummer des freizugebenden Signals (0-31).

SetSignals

AlteSignale = SetSignals(NeueSignale,Maske)
D0 D0 D1

Offset: -306

Beschreibung: SetSignals() überträgt den Zustand derjenigen Signale, deren zugehörige Bits in der Maske gesetzt sind, aus NeueSignale in die Task-Signale (tc_SigRecvd). Ist ein Bit in NeueSignale und der Maske gleich 1, wird das zugehörige Task-Signal gesetzt. Ist es in NeueSignale 0 und die Maske 1, wird es gelöscht. Ist ein Maskenbit = 0, bleibt das entsprechende Task-Signal unbeeinflusst.

Parameter:

NeueSignale enthält den neuen Zustand der Signale

Maske bestimmt, welche Signalbits geändert werden.

Ergebnis:

AlteSignale gibt den alten Zustand der Task-Signale wieder.

SetSignals(0L,0L) liefert die Signale, ohne sie zu verändern.

Signal

Signal(Task, Signale)
 A1 D0

Offset: -324

Beschreibung: Mit dieser Funktion kann man die Signale eines anderen Tasks setzen. Diese Funktion ist die Kernfunktion des Signalsystems, denn mit ihr ist die Signalübermittlung von Task zu Task möglich. Hat der Empfänger-Task auf eines der gesendeten Signale gewartet, kehrt er wieder zum Ready oder Running State zurück. Diese Funktion wird hauptsächlich von dem Message-System benutzt, das im nächsten Kapitel besprochen wird.

Parameter:

Task Zeiger auf die Task-Struktur des Empfänger-Tasks.

Signale eine Signalmaske, die die zu übermittelnden Signalbits enthält.

Wait

Signale= Wait(Signalmaske)
 D0 D0

Offset: -318

Beschreibung: Wait wartet auf die Signale in der angegebenen Signalmaske. Dies bedeutet, daß der Task solange in den Waiting State versetzt wird, bis eines der Signale von einem anderen Task oder einem Interrupt gesetzt wird. War eines der Signale schon vor Aufruf der Wait-Funktion gesetzt, kehrt Wait sofort wieder ins Programm zurück. Als Ergebnis übergibt Wait eine Signalmaske, die alle aufgetretenen Signale enthält, auf die gewartet wurde.

Achtung: Diese Funktion darf nur im USER-Mode aufgerufen werden.

Parameter:

Signalmaske Auf die Signale in dieser Signalmaske wird gewartet.

Ergebnis:

Signale Dies sind diejenigen Signale aus der Signalmaske, die empfangen wurden.

2.4.2.2 Das Message-System

Die Task-Signale bilden die Grundlage für ein weiteres Kommunikationssystem zwischen Tasks, dem sogenannten Message-System. Dieses erlaubt nicht nur die Übergabe von Signalen, sondern auch das Senden und Empfangen von Mitteilungen, die beliebige Daten enthalten können. Auch die automatische Bildung von Warteschlangen, falls der Empfänger nicht schnell genug auf eine Nachricht reagieren kann, ist bei diesem System implementiert. Als Basis für diese Art der Kommunikation dient ein sogenannter Message-Port. Dies ist wieder eine Datenstruktur. In C hat sie folgendes Format (exec/ports.h):

```
struct MsgPort {
    struct Node mp_Node;
    UBYTE mp_Flags;           /*(14) Flags für Aktionsmodus */
    UBYTE mp_SigBit;          /*(15) Signalbit des Tasks */
    struct Task *mp_SigTask; /*(16) Zeiger auf Empfänger-Task */
    struct List mp_MsgList; /*(20) Listenkopf der messagelist*/
};
```

Ein Message-Port ist eine Sammelstelle für Nachrichten (engl. Messages) an einen Task (oder Software-Interrupt). Jeder Task kann Daten an einen Message-Port senden, aber nur ein Task wird von den angekommenen Mitteilungen informiert.

Die einzelnen Felder einer Message-Port-Struktur haben folgende Bedeutung:

mp_Node

Dies ist eine Node-Struktur, wie sie in Kapitel 1 vorgestellt wurde. In ihrem `In_Name` Feld wird wieder ein Zeiger auf den Namen des Message-Ports gespeichert. Dies erleichtert das Auffinden eines bestimmten Message-Ports.

Der Node-Typ in `In_Typ` ist bei einem Message-Port immer `NT_MSGPORT`.

Die anderen Felder der Node-Struktur werden nur verwendet, wenn man einen Message-Port in eine Liste aufnehmen will. Dies kann entweder eine private Liste sein oder die Liste der sogenannten Public-Ports. Dies ist die Liste aller Message-Ports, die Exec bekannt sind.

mp_Flags

Die unteren beiden Bits in diesem Feld bestimmen, was passiert, wenn der Message-Port eine Nachricht empfängt. Es stehen folgende Möglichkeiten zur Auswahl (die entsprechenden Bitkombinationen sind ebenfalls im Include-File "exec/ports.h" enthalten):

- | | |
|-----------------------------|---|
| <code>PA_IGNORE (2)</code> | Diese Kombination der Flagbits bestimmt, daß gar nichts geschieht, wenn eine Nachricht empfangen wird. |
| <code>PA_SIGNAL (0)</code> | Jedesmal, wenn der Message-Port eine Nachricht empfängt, wird das Signal aus dem Feld <code>mp_SigBit</code> an den Ziel-Task gesandt. |
| <code>PA_SOFTINT (1)</code> | Bei jeder empfangenen Nachricht wird ein Software-Interrupt ausgelöst. Näheres über Software-Interrupts können Sie im zugehörigen Kapitel erfahren. |

mp_SigBit

In diesem Feld ist die Nummer des Signalbits gespeichert, das an den Task gesendet wird, wenn `mp_Flags = PA_SIGNAL` ist. Es handelt sich dabei um eine Signalnummer zwischen 0 und 31, es kann immer nur ein Signalbit von einem Message-Port beeinflußt werden.

mp_SigTask

Dieses Feld muß einen Zeiger auf die Task-Struktur des Tasks enthalten, dem das Signal in `mp_SigBit` übermittelt werden soll.

Ist als Modus `PA_SOFTINT` gewählt, enthält `mp_SigTask` stattdessen einen Zeiger auf die Interrupt-Struktur des entsprechenden Software-Interrupts.

mp_MsgList

Dies ist der Listenkopf für die Liste aller eingetroffenen Nachrichten (Messages). Jede empfangene Nachricht wird an das Ende dieser Liste angehängt, und je nach Modus in `mp_Flags` löst dies entweder nichts (`PA_IGNORE`), einen Software-Interrupt (`PA_SOFTINT`) oder ein Signal an den Empfänger-Task (`PA_SIGNAL`) aus. Dieser Listenkopf muß entsprechend initialisiert werden. Dies kann z.B. mit `NewList()` geschehen.

Aufbau einer Nachricht

Jede Nachricht besteht aus einer Message-Struktur und einer beliebigen Nachricht, deren Länge maximal 64 KByte betragen darf. Die Nachricht wird direkt an die Messagestruktur angehängt. Diese Struktur ist ebenfalls in dem Include-File "exec/ports.h" untergebracht:

```
struct Message {
    struct Node  mn_Node;
    struct MsgPort *mn_ReplyPort; /* (14) Antwortport */
    UWORD mn_Lenght; /* (18) Länge der message in Bytes */
};
```

mn_Node

ist eine normale Node-Struktur. Sie dient dazu, die Message in die Liste der empfangenen Messages des Message-Ports einzugliedern. Das *ln_Typ* Feld ist auf den Nodetyp *NT_MESSAGE* zu setzen. Ob man seiner Message einen Namen geben will, bleibt einem selbst überlassen.

mn_Lenght

enthält die Länge der Nachricht in Bytes. Wie schon erwähnt, schließt diese sich direkt an das *mn_Lenght* Feld an.

Senden einer Nachricht

Eine Nachricht wird mittels der *PutMsg*-Funktion an einen beliebigen Message-Port gesendet.

```
PutMsg(Messageport, Message)
```

sendet die Message zu dem Message-Port. Beide müssen als Zeiger auf die zugehörigen Strukturen angegeben werden. Folgendes Beispiel sendet einen Text als Message zu einem hypothetischen Message-Port:

```
{
extern APTR Port; /* Zeiger auf den Message-Port */
static char text[] = "Dies ist eine Beispielnachricht!";
static struct {
    struct Message msg;
    char inhalt[sizeof(text)];
} nachricht;

nachricht.msg.mn_Node.ln_Type=NT_MESSAGE;
strcpy(nachricht.inhalt,text);

PutMsg(Port,&nachricht);
}
```

Beim Senden einer Message wird diese lediglich an die Liste der empfangenen Messages, die *mp_MsgList*, angehängt. Dies geschieht wie üblich mit den *ln_Succ*- und *ln_Pred*-Feldern in

der Node-Struktur der Message, `mn_Node`. Die Message wird nicht kopiert! Dies bedeutet, daß die gesamte Message weiterhin Teil des Tasks ist, der sie gesendet hat. Das Senden einer Message erlaubt also dem Empfänger-Task, einen Speicherbereich des Sender-Tasks zu benutzen.

Empfangen einer Nachricht

Das Empfangen von Nachrichten besteht gewöhnlich aus zwei Schritten. Erst wartet man auf das Signal des Message-Ports, und wenn dieser signalisiert, daß er eine Nachricht empfangen hat, holt man sie dort ab.

Vorausgesetzt, daß der Message-Port ordentlich initialisiert ist, gibt es zwei Möglichkeiten für einen Task, auf die Ankunft einer Message an dem Message-Port zu warten:

Entweder verwendet er die `Wait()`-Funktion oder `WaitPort()`.

```
Message = WaitPort(Port);
```

Diese Funktion wartet auf die Ankunft einer Message an dem Message-Port "Port". Sind dort schon eine oder mehr Messages vorhanden, kehrt `WaitPort()` gleich ins Programm zurück. Ansonsten versetzt sie, wie die `Wait` Funktion, den Task in den Waiting State, bis eine Message ankommt. Als Ergebnis erhält der Task bei `WaitPort()` einen Zeiger auf die Message-Struktur der ersten Message, die Message wird aber nicht vom Message-Port entfernt.

Wann soll man `Wait()` oder `WaitPort()` verwenden?

`Wait()` hat den Vorteil, daß es damit möglich ist, auf mehrere Signale zu warten. Damit ist diese Funktion in Fällen, wo man auf mehrere Ereignisse warten will, am geeignetsten. Will man aber wirklich nur auf eine Nachricht an einem bestimmten Message-Port warten, ist `WaitPort()` günstiger. Denn diese Funktion wartet ja nur, wenn der Message-Port leer ist. `Wait()` hingegen richtet sich nach den Signalen. Hat beispielsweise der

Message-Port schon zwei Messages vor dem Wait() empfangen, tritt folgendes Problem auf:

Der erste Wait()-Aufruf kehrt sofort zurück, da die beiden Messages schon das entsprechende Signal gesetzt haben. Will man jetzt mit einem Wait() auf die nächste Nachricht warten, kann die zu einem Warten ohne Ende werden, denn die gewünschte Message ist ja schon lang vorher angekommen. Die Ursache dieses Problems liegt darin, daß ein Signal nur einmal gesetzt wird, auch wenn mehrere Messages angekommen sind. Man muß daher vor dem zweiten Wait() testen, ob die nächste Nachricht nicht doch schon angekommen ist. Aus diesem Grund ist es besser, WaitPort() zu verwenden, wenn man nur auf einen Message-Port wartet.

Um eine Message von einem Port zu holen, gibt es die GetMsg Funktion:

```
Message = GetMsg(Port);
```

Diese Funktion holt die erste Nachricht von dem angegebenen Port und übergibt einen Zeiger auf ihre Messagestruktur. Die Message wird dann aus der Liste des Message-Ports entfernt. GetMsg() holt die Message an der ersten Position in dieser Liste. Da neue Messages hinten angehängt werden, stellt die Liste der empfangenen Messages eine sogenannte FIFO-Warteschlange dar. FIFO heißt 'First In First Out' und bedeutet, daß das Element, welches zuerst in die Liste aufgenommen wurde, auch zuerst wieder ausgegeben wird.

Also kann man mit GetMsg() der Reihe nach alle empfangenen Messages vom Port holen. Sind keine weiteren Messages mehr vorhanden, kehrt GetMsg() mit einer 0 zurück.

Folgendes Beispiel benutzt WaitPort() und GetMsg(), um eine Nachricht aus einem hypothetischen Port zu holen:


```
extern struct MsgPort *Port;
struct Message *GetMsg();
int signal;

if((signal=AllocSignal(-1L))<0)
    {printf("Keine freien Signale mehr übrig!");return(0);}

Port->mp_Flags=PA_SIGNAL;
Port->mp_SigBit=sig;
Port->mp_SigTask=FindTask(0); /* eigener Task */

WaitPort(Port);
message = GetMsg(Port);
```

Antworten auf eine Nachricht

Hat ein Task eine Nachricht gesendet, will er natürlich auch wissen, ob sie angekommen ist. Grund dafür ist, daß die gesamte Message, einschließlich der Messagestruktur, ja weiterhin zu seinem Task gehört. Indem er sie sendet, gibt er dem Empfänger die Erlaubnis, den Speicherbereich, in dem die Message steht, auszulesen. Außerdem kann der Empfänger noch irgendwelche Rückmeldungen oder Ergebnisse in der Message unterbringen. Da der Sender den Speicherbereich meistens wieder für andere Zwecke, z.B. eine neue Nachricht, verwenden will, muß er wissen, wann der Empfänger die Nachricht empfangen und verarbeitet hat. Er kann sie ja nicht einfach löschen, ohne zu wissen, ob sie schon gelesen wurde. Aus diesem Grund wurde ein sogenannter Reply-Port (Antwort-Port) eingerichtet. Ein Reply-Port kann ein beliebiger Port des Sender-Tasks sein. Um dem Empfänger die Adresse dieses Ports mitzuteilen, gibt es noch ein Feld in der Message-Struktur, das wir noch nicht erwähnt haben:

mn_ReplyPort

enthält die Adresse der Reply-Ports des Sender-Tasks. Um eine Nachricht zu beantworten, sendet der Empfänger sie einfach an diesen Reply-Port zurück, nachdem er sie gelesen und eventuell verändert hat. Der Sender weiß damit, daß die Message empfan-

gen und bearbeitet wurde. Er kann jetzt den Speicher der Message neu verwenden oder einfach dem System zurückgeben.

```
ReplyMsg(Message);
```

erledigt dieses Rücksenden der Message.

Schaffen neuer Message-Ports

Um einen Message-Port zu schaffen, muß man eigentlich nur eine korrekt initialisierte Message-Port-Struktur im Speicher unterbringen. Am Einfachste geschieht dies durch eine C-Struktur mit der Speicherklasse static. Allerdings kann man sich auch den nötigen Speicher mittels AllocMem() vom System holen und initialisieren.

Hat man eine fertige Message-Port-Struktur, muß man sich noch entscheiden, ob man sie zur Liste der öffentlichen Message-Ports hinzufügen will. Dies kann mit der AddPort Funktion geschehen. Diese Funktion übernimmt auch das Initialisieren des mp_MsgList Felds der Messagestruktur und macht so einen Aufruf von NewList() überflüssig. AddPort benötigt als einziges Parameter die Adresse der Messagestruktur. Der Vorteil eines Message-Ports in der Liste der öffentlichen Ports ist, daß ein anderer Task diesen Port schnell anhand seines Namens auffinden kann. Fügt man einen Port nicht zu dieser Liste hinzu, muß man jedem Task, der mit ihm kommunizieren soll, die Adresse des Message-Ports mitteilen.

Um einen Port in der Liste anhand seines Namens aufzufinden, gibt es die FindPort-Funktion.

```
Port=FindPort(Name);
```

sucht einen Message-Port mit dem angegebenen Namen und übergibt, falls er ihn findet, dessen Adresse. Fügt man einen Port mit AddPort() zum System hinzu, sollte man vorher überprüfen, ob nicht schon ein Port dieses Namens vorhanden ist.

Benötigt man einen Port nicht mehr, kann man ihn einfach löschen, nachdem man auf alle noch ausstehenden Messages gewartet und diese mit ReplyMsg() bantwortet hat.

Gehört der Message-Port zu den öffentlichen Ports, muß man ihn mit RemPort(Port) aus dem System entfernen, bevor man ihn löscht (bzw. seinen Speicher wieder freigibt).

Um das Erstellen neuer Message-Ports zu vereinfachen, gibt es die CreatePort Funktion. Diese Funktion gehört nicht zum Betriebssystem, sonder findet sich im Runtime-Library eines Amiga C-Compilers (amiga.lib). Ihr C-Source-Code lautet wie folgt:

```
#include <exec/exec.h>

extern APTR AllocMem();
extern UBYTE AllocSignal();
extern struct Task *FindTask();

struct MsgPort *CreatePort (Name, Pri)
char *Name;
BYTE Pri;
{
    BYTE Signal;
    struct MsgPort *Port;

    if ((Signal=AllocSignal(-1))==-1)
        return((struct MsgPort *)0);

    Port=AllocMem((ULONG)sizeof(*Port),MEMF_CLEAR|MEMF_PUBLIC);

    if (Port==0) {
        FreeSignal (Signal);
        return((struct MsgPort *)0);
    }

    Port->mp_Node.ln_Name = Name;
    Port->mp_Node.ln_Pri = Pri;
    Port->mp_Node.ln_Type = NT_MSGPORT;

    Port->mp_Flags = PA_SIGNAL;
    Port->mp_SigBit = Signal;
    Port->mp_SigTask = FindTask(0);
```

```
if (name != 0)
    AddPort(Port);
else
    NewList (&(Port->mp_MsgList));

return(Port);
}
```

Diese Funktion schafft einen Message-Port mit dem angegebenen Namen und angegebener Priorität. Als Ergebnis erhält man die Adresse des neuen Ports oder 0, wenn kein Speicher oder kein Signal mehr frei war. Ist der Zeiger auf den Namen ungleich Null, wird der Port der Liste der öffentlichen Ports hinzugefügt. Mit dieser Funktion kann man z.B schnell und einfach einen ReplyPort aufbauen.

Auch zum Löschen eines Ports gibt es eine Funktion im *amiga.lib*:

```
DeletePort(Port)
    struct MsgPort *Port;
{
    if ((Port->mp_Node.ln_Name) != 0)
        RemPort(Port);

    Port->mp_Node.ln_Type = 0xFF;
    Port->mp_MsgList.lh_Head=(struct Node *)-1;

    FreeSignal(Port->mp_SigBit);

    FreeMem(Port,(ULONG) sizeof(*Port));
}
```

DeletePort() löscht den angegebenen Port. Hat er einen Namen, wird er auch aus der Liste der öffentlichen Ports gelöscht.

Task-Exceptions (Ausnahmestände)

Es ist manchmal unbefriedigend, daß beim Warten auf ein Signal der Task nicht weiterlaufen kann. Nehmen wir einmal an, ein Task zeichnet eine mathematische Funktion. Da dies sehr lange dauert, soll die Möglichkeit bestehen, den Zeichenvorgang durch Tastendruck abubrechen. Dieser soll über einen Message-Port übermittelt werden. Nun müßte man innerhalb der Zeichenschleife ständig die Signale testen, ob der Port eine Message empfangen hat. Damit würde man aber die Zeichenschleife verlangsamen. Mit diesem unbefriedigenden Zustand muß man sich aber beim Amiga nicht abfinden. Es gibt da nämlich die sogenannte Task-Exception (ein Ausnahmestand des Tasks).

Ähnlich einem Interrupt, wird der Task durch das Auftreten eines Signals unterbrochen. Dies kann an jeder beliebigen Stelle geschehen. Dann wird der sogenannte Exception Handler aufgerufen. Dieser ist Teil des Ursprungs-Tasks und hat deshalb Zugriff auf dessen Daten (vorausgesetzt, sie sind nicht lokal). In unserem Beispiel könnte er die Schleifenvariable auf den Endwert setzen und damit das Zeichnen beenden. In Maschinensprache hat man noch mehr Möglichkeiten. Man kann z.B. den Task direkt manipulieren.

Um eine Task-Exception zu ermöglichen, sind folgende Schritte notwendig:

1. Die Startadresse des Exception Handlers muß im zugehörigen Feld der Task-Struktur, `tc_ExceptCode`, untergebracht werden. Es besteht die Möglichkeit, auch noch einen Zeiger auf gemeinsame Daten in `tc_ExceptData` zu schreiben.
2. Es muß festgelegt werden, welche Signale eine Exception auslösen dürfen. Das `tc_SigExcept`-Feld in der Task-Struktur bestimmt dies. Jedes dort gesetzte Signal löst eine Exception aus, wenn es vom Task empfangen wird. Um das Setzen und Löschen der Bits in diesem Feld zu vereinfachen, gibt es eine spezielle Funktion: `SetExcept`. Ihre

genaue Beschreibung findet man in der Funktionsübersicht am Ende dieses Kapitels.

Tritt eine Exception auf, legt Exec zuerst die aktuellen Inhalte der Prozessorregister (PC, SR, D0-D7 und A0-A6) auf den Task-Stack, um eine störungsfreie Fortsetzung des Tasks nach dem Ende der Exception zu ermöglichen.

Dann kommt eine Signalmaske in D0, die alle aufgetretenen Exception-Signale enthält. Die Adresse im tc_ExceptData Feld wird nach A1 kopiert. Anschließend beginnt die Abarbeitung des Exception Codes an der Adresse in tc_ExceptCode.

Am Ende einer Exception muß ein RTS stehen. Dann holt Exec den alten Registerinhalt wieder vom Stack und fährt mit dem Task fort.

Während einer Exception verhindert Exec das Auftreten weiterer Exceptions. Um nach dem Ende einer Exception ihr erneutes Auftreten zu erlauben, muß man in D0 denselben Wert zurückgeben, der einem dort am Anfang übergeben wurde, d.h. man ändert einfach nicht den Inhalt von D0.

Tritt während einer Exception ein Signal auf, das eine neue Exception auslöst, wird diese nach dem Ende der gerade ablaufenden erneut gestartet.

War ein Signal schon gesetzt, bevor man ihm mittels SetExcept erlaubte, eine Exception auszulösen, tritt diese sofort auf.

Prozessor Traps

Eine andere Art von Ausnahmeständen sind die Traps. Mit ihnen sind bestimmte Ausnahmestände des 68000er Prozessors gemeint. Diese werden bei Motorola (dies ist die Herstellerfirma des 68000) Exceptions genannt, was man nicht mit oben beschriebenen Task Exceptions verwechseln darf. Folgende 68000 Exceptions werden von Exec als Traps bezeichnet:

Traps:

- | | |
|-------|--|
| 2 | Busfehler (Bus error) |
| 3 | Adressfehler (Address error) |
| 4 | Illegaler Befehl (Illegal instruction) |
| 5 | Division durch null (Zero divide) |
| 6 | CHK Befehl (CHK instruction) |
| 7 | TRAPV Befehl (TRAPV instruction) |
| 8 | Privilegverletzung (Privilege violation) |
| 9 | Ablaufverfolgung (Trace) |
| 10 | Befehl mit 1010 am Anfang (Line 1010 emulator) |
| 11 | befehl mit 1111 am Anfang (Line 1111 emulator) |
| 32-37 | Trap-Befehle (Trap instructions) |

Ein Trap ist immer eine direkte Folge einer Anweisung im Programm. Dies kann entweder gewollt (CHK, TRAP, TRAPV, 1010, 1111 oder Trace) oder aufgrund eines Programmfehlers (Bus- oder Adressfehler, Division durch Null oder Privilegverletzung) passieren.

Tritt also ein Prozessor-Trap auf, springt Exec in einen sogenannten Trap-Handler. Die Adresse dieses Handlers wird aus dem `tc_TrapCode` Feld geholt. Normalerweise befindet sich dort ein Zeiger auf den Standard-Trap-Handler von Exec. Dieser erzeugt den (leider) nur allzu gut bekannten "Software Error - Task held"-Requester oder gar eine Guru Meditation.

Man kann aber die Adresse in `tc_TrapCode` auf einen eigenen Handler umbiegen. Dieser Handler kann dann entweder auf alle Traps reagieren oder nur einige bestimmte behandeln und ansonsten in den Standard-Handler weiterspringen.

Ein Trap-Handler wird beinahe direkt angesprungen. Exec legt lediglich noch die Trap-Nummer (siehe obige Liste) auf den Stack. Dies hat folgende Auswirkungen:

Erstens befindet man sich im Supervisor-Modus und arbeitet mit dem Supervisor-Stack! Während des Ablaufs des Trap-Handlers ist daher das Task-Switching gesperrt.

Zweitens kann der Inhalt des Stacks variieren. Normalerweise hat er folgendes Format:

Stackpointer (SSP)	Trap Nummer (Langwort)
Stackpointer +4	Statusregister (Wort)
Stackpointer +2	Rücksprungadresse (Langwort)

Bei einem Adreß- oder Busfehler werden aber weitere Informationen auf den Stack gelegt. Obendrein sind diese bei neueren Vertretern der 68xxx Familie, wie dem 68010 und 68020, wieder anders. Um vollständige Kompatibilität zu wahren, muß man also einiges beachten. Am besten nehmen Sie ein gutes 68000-Buch zu Hilfe, um alle Möglichkeiten einzuschließen.

Oder man springt bei einem Adreß- bzw. Busfehler in den Trap-Handler von Exec, da bei diesen Ausnahmen meist doch nichts mehr zu machen ist.

Um den Trap-Handler wieder zu verlassen, nimmt man die Trap Nummer vom Stack (Achtung: Langwort) und verläßt ihn mit einem RTE-Befehl. Da keine zusätzlichen Register von Exec auf dem Stack gesichert werden, darf man den Inhalt der Prozessorregister nicht dauerhaft verändern!

Folgendes Beispiel verwendet einen Trap-Handler zum Auffangen eines "Division durch Null"-Fehlers. Da man einen Trap-Handler schlecht in C schreiben kann, wurde er mittels `#asm` und `#endasm` direkt in Maschinsprache im Sourcecode integriert. Da nicht alle C-Compiler diese Preprozessoranweisungen kennen (das Programm wurde mit Aztec C erstellt), muß man bei anderen Produkten C- und Maschinspracheteil getrennt compilieren bzw. assemblieren und anschließend gemeinsam linken.


```
/**** Auffangen einer 68000er Exception ****/

#include <exec/execbase.h>

extern struct ExecBase *SysBase;

main()
{
    /*** Hinzufügen des Trap-Handlers ***/

    extern APTR Trap;
    APTR oldtrap;
    USHORT zahl1,zahl2;
    struct Task *ThisTask;

    oldtrap=SysBase->ThisTask->tc_TrapCode;
    SysBase->ThisTask->tc_TrapCode=&Trap;
    SysBase->ThisTask->tc_TrapData=(APTR)0;

    /*** Auslösen eines "Division by zero" Traps ***/

    zahl1 = 10; zahl2 = 0;
    zahl1 = zahl1/zahl2;

    if((ULONG)SysBase->ThisTask->tc_TrapData==0)
        printf("Diese Stelle wird nie erreicht!");
    else
        printf
            ("Exception erkannt, da tc_TrapData = Trap-Nummer:%ld\n",
             SysBase->ThisTask->tc_TrapData);

    /*** Trap-Handler wieder entfernen ***/

    SysBase->ThisTask->tc_TrapCode=oldtrap;

}

/*** Trap-Handler ***/
/** Nur mit Aztec-C sicher lauffaehig **/
/** TAB vor Opcode ist notwendig! **/
```

```
#asm

_Trp move.l a0,-(sp)
      move.l 4,a0          ;SysBase
      move.l 276(a0),a0    ;SysBase->ThisTask
      move.l 4(sp),46(a0) ;Trap-Nummer nach
                          ;SysBase->ThisTask->tc_TrapData

      move.l (sp),a0
      add.l #8,sp
      rte

#endasm
```

Ohne den Trap-Handler würde dieses Programm mit einem "Software Error - Task held" abstürzen, da `zahl1` durch 0 dividiert wird. Als Beweis dafür, daß dies wirklich einen Trap ausgelöst hat, dient der `if`-Befehl. Denn nur der Trap kann `tc_TrapData` ungleich 0 setzen, nachdem es gerade vorher vom Task gelöscht wurde. Aber direkt nach der illegalen Division springt Exec in den Trap-Handler. Dieser nimmt die Trap-Nummer vom Supervisor-Stack und schreibt sie in das `tc_TrapData`-Feld. Daher gibt der `printf()`-Aufruf auch die Zahl 5 auf dem Bildschirm aus, denn dies ist die Trap-Nummer einer Division durch Null.

Die Trap-Befehle

Auch ein Trap, der durch einen der 16 Trap-Befehle ausgelöst wurde (Trap-Nummern 32 bis 47), springt in den Trap-Handler. Allerdings gibt es bei den Traps die Möglichkeit, bestimmte Trap-Nummern vorher zu belegen, ähnlich wie bei den Signalen mittels einer `AllocTrap` und einer `FreeTrap`-Funktion (genaue Erklärung siehe anschließende Funktionsliste).

Das Belegen und Freigeben von Traps dient aber lediglich der Ordnung, damit immer klar ist, welche Traps gerade benutzt werden und welche nicht. Tritt ein Trap-Befehl auf, wird immer in den aktuellen Trap-Handler verzweigt, unabhängig davon, ob ein Trap-Befehl mit `AllocTrap` belegt wurde oder nicht.

Funktionen des Message-Systems, der Traps und Exceptions**AddPort****AddPort(Port)****A1****Offset:** -354

Beschreibung: AddPort() fügt die angegebene Message-Port-Struktur in die Liste der öffentlichen Ports ein. Sie wird dort nach ihrer Priorität eingereiht. Der Listheader dieser Liste kann über SysBase->PortList angesprochen werden. AddPort() initialisiert auch die mp_MsgList-Struktur innerhalb des Message-Ports.

Parameter:

Port Zeiger auf eine Message-Port-Struktur

AllocTrap**Trapnummer = AllocTrap(Trapnummer)****D0****D0****Offset:** -342

Beschreibung: AllocTrap besetzt einen der Trap-Befehle des 68000. Man kann als Trap-Nummer eine Zahl zwischen 0 und 15 angeben, um den entsprechenden Trap-Befehl zu besetzen, oder -1, dann sucht AllocTrap den nächsten freien Trap-Befehl.

Parameter:

Trapnummer Eine Zahl zwischen 0 und 15 für einen bestimmten Trap-Befehl, oder -1 für den ersten unbesetzten.

Ergebnis:

Trapnummer enthält den tatsächlich besetzten Trap-Befehl oder -1 als Zeichen dafür, daß der gewünschte Trap-Befehl nicht frei war bzw. daß überhaupt kein Trap-Befehl mehr frei war.

FindPort

```
Port = FindPort(Name)
D0          A1
```

Offset: -390

Beschreibung: FindPort() sucht den nächsten Message-Port mit dem angegebenen Namen in der Liste der öffentlichen Ports und gibt, falls ein Port mit dem angegebenen Namen existiert, einen Zeiger auf diesen Port zurück.

Parameter:

Name Der Name des zu suchenden Ports

Ergebnis:

Port Ein Zeiger auf einen Message-Port mit dem angegebenen Namen oder Null, wenn ein solcher nicht existiert.

FreeTrap

```
FreeTrap(Trapnummer)
D0
```

Offset: -348

Beschreibung: FreeTrap gibt den Trap-Befehl mit der angegebenen Nummer wieder frei.

Parameter:

Trapnummer Die Nummer des Trap-Befehls (0 bis 15)

PutMsg

PutMsg(Port,message)
A0 A1

Offset: -366

Beschreibung: PutMsg sendet eine Nachricht an einen Message-Port. Dort wird sie an die Liste der empfangenen Nachrichten angehängt, und die dem Inhalt des mp_Flags-Feld entsprechende Aktion wird ausgelöst.

Parameter:

Port Die Adresse der Message-Port-Struktur des Zielports

message Zeiger auf die Messagestruktur der Message

RemPort

RemPort(Port)
A1

Offset: -360

Beschreibung: Diese Funktion entfernt einen Message-Port aus der Liste der öffentlichen Ports. Es ist dann nicht mehr möglich, ihn mittels FindPort anzusprechen.

Parameter:

Port Zeiger auf den Message-Port

ReplyMsg

ReplyMsg(message)
A1

Offset: -378

Beschreibung: ReplyMsg() sendet eine Message zurück zu ihrem Reply-Port. Ist das mn_ReplyPort Feld der Messagestruktur gleich null, wird diese Funktion ignoriert.

Parameter:

message Zeiger auf die Messagestruktur

SetExcept

AlteSignale= SetExcept(NeueSignale, Maske)
D0 D0 D1

Offset: -312

Beschreibung: SetExcept bestimmt, welche Signale eine Exception auslösen können. Das genaue Verhalten dieser Funktion entspricht demjenigen von SetSignals().

Parameter:

NeueSignale sind die neuen Zustände der Exception-Signale

Maske Die Maske bestimmt, welche Exception-Signale überhaupt beeinflusst werden sollen.

Ergebnis:

AlteSignale sind die Zustände der Exception-Signale vor der Änderung

WaitPort

```
message = WaitPort(Port)
                        A0
```

Offset: -384

Beschreibung: WaitPort wartet auf den Empfang einer Message an einem bestimmten Port. Trifft dort eine Message ein, oder war schon vor dem Aufruf von WaitPort() eine Message vorhanden, kehrt WaitPort mit der Adresse dieser Message zurück. Die Message wird aber nicht aus der Liste der empfangenen Messages gelöscht. Dazu muß GetMsg() verwendet werden.

Parameter:

Port Adresse des Ports

Ergebnis:

message Zeiger auf die erste Message in der Liste

2.5 Speicherverwaltung des Amiga

Der Amiga verfügt über eine dynamische Speicherverwaltung, was bedeutet, daß Bildschirmspeicher, Diskspeicher usw. sowie die zu ladenden Programme in keinem bestimmten, sich nie ändernden Speicherbereich geladen werden, sondern bei jedem Laden einen anderen Bereich zugewiesen bekommen können. Diese dynamische Speicherverwaltung macht es möglich, daß mehrere Programme gleichzeitig im Speicher abgearbeitet werden können, da kein Programm auf einen bestimmten Speicherbereich angewiesen ist, wie es bei anderen Computern, beispielsweise dem C64, der Fall ist.

Dem System muß lediglich mitgeteilt werden, daß Speicher gebraucht wird, worauf dieses ihn, sofern er noch frei ist, dem Anwender zuweist. Vom System wird nicht gemerkt, welches Programm (Task) welchen Speicher belegt hat, sondern nur vermerkt, daß er für andere Tasks nicht mehr zur Verfügung steht.

Genau gesagt: es wird nicht vermerkt, welcher Speicher besetzt, sondern umgekehrt, welcher noch erreichbar ist.

Wenn ein Task einen bestimmten Speicherbereich nicht mehr benötigt, sollte er dies dem System mitteilen, damit der Speicher einem anderen Task zuteilen kann. Wird der nicht benötigte Speicher nicht an das System zurückgegeben, bleibt er belegt, bis ein Reset erfolgt. Die Folge ist natürlich eine drastische Verringerung der Speicherkapazität.

Wenn Speicher belegt werden soll, ist dies nur in Acht-Byte-schritten möglich. Anderenfalls rundet das System die angegebene Speichergröße bis zum nächsten Acht-Byte-Schritt auf. Somit ergibt sich eine minimale Bereichszuweisung von acht Bytes.

Zum Belegen und Freigeben des gewünschten Speicherbereichs existieren in der Exec-Library mehrere Funktionen, von denen zwei am häufigsten Verwendung finden. Die Funktionen sind AllocMem() und FreeMem().

Bei der Belegung von Speicher müssen Sie dem System mitteilen, welche Art von Speicher Sie zugewiesen haben möchten und ob dieser noch besondere Eigenschaften haben soll.

Diese Bedingungen, die gewählt werden können, sind:

MEMF_CHIP

gibt an, daß der zuzuweisende Speicherbereich Chip-Memory sein muß. Chip-Memory ist der untere 512KB-Bereich, der von Chips wie z.B. dem Blitter angesprochen werden kann. Grafik, Sound usw. muß in diesem Bereich abgelegt werden. Auch wenn Sie zur Zeit nur 512KB Speicher besitzen und somit diese Bedingung immer erfüllt wird, sollten Sie aus Kompatibilitätsgründen zu Geräten mit größerem Speicher nicht auf die genaue Bestimmung verzichten. Der Code für diese Bedingung ist \$02 und wird, ebenso wie auch die anderen Bedingungen, in C durch das Memory-Include-File, definiert.

MEMF_FAST

gibt an, daß sich der zuzuweisende Speicher außerhalb der unteren 512 KByte befinden muß. Die Zuweisung führt jedoch nur mit der Verwendung einer RAM-Erweiterung zu einem positiven Ergebnis. Der Code der Bedingung ist \$04.

MEMF_PUBLIC

gibt an, daß der zuzuweisende Speicher, nachdem er festgelegt wurde, nicht verschoben werden darf. Diese Verschiebung ist in der jetzigen Version des Betriebssystems nicht integriert, sollte jedoch aus Kompatibilitätsgründen bei der Speicherzuweisung für Tasks, Interrupts, Message-Ports usw. verwendet werden. Der Code für diese Bedingung ist \$01.

MEMF_CLEAR

gibt an, ob der zuzuweisende Speicher für die Übergabe mit Nullen gelöscht werden soll. Der Code hierfür ist \$10000.

MEMF_LARGEST

gibt an, daß der zuzuweisende Speicher der größte zur Verfügung stehende Speicherblock sein soll. Der Code hierfür ist \$20000.

Sollen mehrere Bedingungen wie Chip und Clear erfüllt sein, so müssen die Codes miteinander ODER-verknüpft werden.

Sollte weder die Angabe Chip- oder Fast-Memory erfolgt sein, wird zuerst versucht, Fast-Memory zu belegen. Erst nach dem Mißlingen des Versuchs wird Chip-Memory belegt.

Vorsicht:

Es sollte vermieden werden, aus einem Interrupt Speicher zu belegen oder wieder zurückzugeben, da die Routinen des Amiga, die für diese Tätigkeiten gedacht sind, die Interrupts nicht sperren. Sollte sich ein Task gerade in einer Routine zur Speicherverwaltung befinden und von einem Interrupt unterbrochen

werden, der ebenfalls mit den Speicherroutinen arbeitet, so kann das System in große Schwierigkeiten geraten. Das gleiche gilt auch für den Aufruf von Routinen aus einem Interrupt heraus, die nicht unterbrochen werden dürfen, jedoch nicht die Interrupts abschalten.

2.5.1 Die Funktionen AllocMem() und FreeMem()

AllocMem

Speicher = AllocMem(Speichergöße, Bedingungen)
 D0 D0 D1

Offset: -198

Beschreibung: Die Funktion sucht einen freien Speicherbereich, der den angegebenen Bedingungen entspricht, und kennzeichnet ihn als belegt. In D0 wird die Anfangsadresse des gefundenen Speichers angegeben. Sollte es nicht möglich sein, den gewünschten Speicher zu belegen, wird in D0 als Fehlermeldung eine Null übergeben.

Parameter:

Speichergöße gibt an, wie groß der Speicher ist, der zugewiesen werden soll.

Bedingungen sind die zuvor beschriebenen Bedingungen, die der AllocMem()-Funktion übergeben werden und nach denen der Speicherbereich ausgesucht wird (z.B. Chip-Memory).

Mit der AllocMem()-Funktion kann man sich keinen zuvor genau bestimmten Speicherbereich zuweisen lassen, sondern nur einen gerade zur Verfügung stehenden.

So wie es eine Funktion zum Zuweisen und somit zum Belegen des Speichers gibt, existiert ebenso eine Funktion zum Freigeben des Speichers.

FreeMem

FreeMem(Speicherblock, Größe)

A1

D0

Offset: -210

Beschreibung: Die Funktion gibt den zuvor belegten Speicher wieder dem System zurück und ermöglicht so eine erneute Belegung durch andere Tasks. Die von der Funktion übernommenen Parameter werden gerundet.

Parameter:

Speicherblock ist ein Zeiger auf den Anfang des Speicherbereichs, den man dem System zurückzugeben wünscht. Der Zeiger wird auf das nächste Vielfache von 8 abgerundet.

Größe gibt an, wieviel Speicher man gedenkt freizugeben. Die Größe wird auf das nächste Vielfache von 8 aufgerundet.

Vorsicht:

Sollte versucht werden, bereits freien Speicher erneut als "frei" zu kennzeichnen, führt dies zu einem Absturz mit der Guru-nummer 81000009.

Das folgende C-Programm zeigt, wie man sich Speicher zuweisen und wieder freigeben lassen kann.

```
#include <exec/memory.h>
#include <exec/types.h>

#define SIZE 1000

main()
{
    ULONG Speicher;

    Speicher = AllocMem (SIZE, MEMF_CHIP | MEMF_PUBLIC);
```

```

    if (Speicher = 0) {
        printf ("\n Speicher nicht erreichbar\n");
        exit (0);
    }
    printf ("\n Speicher zugewiesen\n");

    FreeMem (SIZE, Speicher);
}

```

In 'Speicher' wird die Anfangsadresse des Speichers vermerkt, der zugewiesen werden konnte.

2.5.2 Die Memory-List-Struktur

Oft ist es nötig, mehrere verschiedene Speicherbereiche zu belegen. Zu diesem Zweck könnte man für jeden einzelnen Bereich jedesmal die AllocMem()-Funktion aufrufen. Sie sehen sicher ein, daß dies ein recht aufwendiges Unterfangen ist. Aus diesem Grund stellt die Exec-Library zwei Funktionen zur Verfügung, die uns diese Aufgabe erleichtern. Die Funktionen heißen AllocEntry() und FreeEntry().

Um eine der Funktionen aufrufen zu können, muß zuvor, wie könnte es beim Amiga auch anders sein, eine Struktur initialisiert werden, aus der sich die Funktionen ihre Werte nehmen.

Die Struktur heißt MemList und sieht wie folgt aus:

```

struct MemList {
0   struct Node ml_Node;
14   UWORD ml_NumEntries;
16   struct MemEntry ml_ME[1];
};

```

ml_Node

ist eine uns bekannte Node-Struktur, um mehrere MemList-Strukturen miteinander verknüpfen zu können.

ml_NumEntries

gibt an, wieviel Speicherbereiche man zu belegen gedenkt.

ml_ME[1]

ist wiederum eine eigene Struktur, in der eingetragen wird, welche Bedingungen der zu reservierende Speicher hat und wie groß der Speicherbereich sein soll. Die Struktur hat folgendes Aussehen:

```
struct MemEntry {
    union {
        ULONG meu_Reqs;
        APTR meu_Addr;
    } me_Un;
    ULONG me_Length;
};

#define me_un me_Un
#define me_Reqs me_Un.meu_Reqs
#define me_Addr me_Un.meu_Addr
```

me_Un

ist aufgrund der 'union'-Bedingung geteilt. Zum einen können in *me_Un* die Bedingungen (Requests) für die Speicherzuweisung enthalten sein, andererseits den Zeiger auf den reservierten Speicher enthalten. Bei der Erstellung der *MemEntry*-Struktur zum Aufruf der *AllocEntry()*-Funktion befinden sich hier die Bedingungen (z.B. *MEMF_CHIP*) für die Speicherzuweisung, nach dem Aufruf der Funktion jedoch die Anfangsadresse des zugewiesenen Speichers.

me_Length

gibt die Länge des zuzuweisenden Speichers an.

AllocEntry

```
Liste = AllocEntry(MemList)
D0                      A0
```

Offset: -222

Beschreibung: Der Funktion wird ein Zeiger auf eine MemList-Struktur in A0 übergeben. Die Funktion versucht nun, alle in der Struktur eingetragenen Speicherbereiche zu belegen. Ist ein Bereich belegt, wird an der Stelle, wo sich zuvor die Bedingungen (Requirements) befanden, der Zeiger auf den gefundenen Speicher abgelegt.

Parameter:

Liste ist ein Zeiger auf die neuerstellte MemList-Struktur. Sollte ein Fehler während der Zuweisung aufgetreten sein, wird in D0 die Bedingungen des nicht erreichbaren Speichers zurückgegeben, wobei das oberste Bit (Bit 31) gesetzt ist. Es wird in dem Fall kein Speicher belegt, auch wenn andere Entries hätten ausgeführt werden können.

FreeEntry

```
FreeEntry(Liste)
A0
```

Offset: -228

Beschreibung: Die Funktion gibt alle die in der MemList-Struktur vermerkten Speicherbereiche dem System zurück.

Parameter:

Liste ist ein Zeiger auf die MemList-Struktur, die von der AllocEntry()-Funktion übergeben wurde.

Weder mit der AllocEntry()- noch mit der FreeEntry()-Funktion ist es möglich, mehrere verkettete MemList-Strukturen gleichzeitig zu bearbeiten.

Sie werden sich vielleicht fragen, wie es möglich ist, mehrere Entries mit einer MemList-Struktur zu initialisieren, da doch in der MemList-Struktur eindeutig nur eine MemEntry-Struktur eingebunden ist (ml_Me[1]). Um mehrere Entries zu initialisieren, muß zu einem Trick gegriffen werden. Man muß sich eine eigene Struktur erstellen, die beispielsweise folgendes Aussehen hat:

```
struct {  
    struct MemList me_Kopf;  
    struct MemEntry me_Zusatz[3];  
} MeineListe;
```

Man übergibt der AllocEntry-Funktion jetzt nicht den Zeiger auf eine MemList-Struktur, sondern den Zeiger auf seine eigene initialisierte Struktur, mit der es möglich ist, mehrere Entries zu initialisieren. In dem soeben gegebenen Beispiel werden vier Speicherbereiche von der AllocEntry-Funktion belegt, da die MemEntry-Struktur noch über eine eigene MemEntry-Struktur verfügt.

Sehen wir uns die Verwendung der AllocEntry()-Funktion einmal anhand eines Beispiels an.

```
#include <exec/memory.h>  
#include <exec/types.h>  
  
struct MemListe {  
    struct MemList me_Kopf;  
    struct MemEntry me_Zusatz[2]; };  
  
main()  
{  
    struct MemList *Speicherliste, *AllocEntry();  
  
    struct MemListe MeineListe;  
  
    MeineListe.me_Kopf.ml_NumEntries = 3;  
    MeineListe.me_Kopf.ml_me[0].me_Reqs = MEMF_CLEAR;
```

```
MeineListe.me_Kopf.ml_me[0].me_Length = 100;
MeineListe.me_Kopf.ml_me[1].me_Reqs = MEMF_CLEAR | MEMF_FAST;
MeineListe.me_Kopf.ml_me[1].me_Length = 1900;
MeineListe.me_Kopf.ml_me[2].me_Reqs = MEMF_PUBLIC | MEMF_CHIP;
MeineListe.me_Kopf.ml_me[2].me_Length = 300;

Speicherliste = AllocEntry (&MeineListe);

if ( ((ULONG)Speicherliste) >> 30) {
    printf("\n Nicht alle Einträge konnten mit Erfolg
zugewiesen werden\n");
    exit (0);
};
}
```

2.5.3 Speicherzuweisung und Tasks

Wenn von einem Task aus Speicher belegt werden soll, so empfiehlt es sich, dies mit der AllocEntry()-Funktion zu machen. In der Task-Struktur ist nämlich eine List-Struktur eingebunden ist (tc_MemEntry), in die der belegte Speicher in Form von MemList-Struktur in einer Liste verknüpft werden kann. Der Speicher, der in diesen Listen verknüpft ist, kann dann leicht von Ihrer Routine, die den Task auflöst, wieder an das System zurückgegeben werden.

Ein weiterer Vorteil beim Eintragen des verwendeten Speichers in die Liste ist, daß der Task jederzeit nachsehen kann, welche Bereiche er belegt hat.

Es ist natürlich auch möglich, einen Speicherbereich mit der AllocMem()-Funktion zu belegen und diesen daraufhin in eine solche Liste einzutragen.

2.5.4 Die interne Verwaltung des Speichers

Nachdem jetzt bekannt sein dürfte, wie man für seine Zwecke Speicherplatz reserviert, wollen wir uns einmal ansehen, wie der Speicher intern verwaltet wird.

Wie man sich denken kann, wird die Verwaltung des Speichers wieder über eine Struktur erledigt. Diese Struktur sieht wie folgt aus:

```
struct MemHeader {
0   struct Node mh_Node;
14  UWORD mh_Attributes;
16  struct MemChunk *mh_First;
20  APTR mh_Lower;
24  APTR mh_Upper;
28  ULONG mh_Free;
};

#define MEMF_PUBLIC (1L<<0)
#define MEMF_CHIP   (1L<<1)
#define MEMF_FAST    (1L<<2)
#define MEMF_CLEAR   (1L<<16)
#define MEMF_LARGEST (1L<<17)
#define MEM_BLOCKSIZE 8L
#define MEM_BLOCKMASK 7L
```

mh_Node

ist wieder unsere bekannte Node-Struktur, um die MemHeader-Struktur in einer List-Struktur verknüpfen zu können.

mh_Attributes

gibt an, welche Bedingungen der verwaltete Speicher hat. z.B. MEMF_FAST

**mh_First*

ist ein Zeiger auf die erste MemChunk-Struktur. Der Aufbau und der Sinn dieser Struktur werden noch besprochen.

mh_Lower

ist ein Zeiger auf den Speicheranfang, der von dem Header verwaltet wird.

mh_Upper

ist ein Zeiger auf das Speicherende, das von dem Header verwaltet wird.

mh_Free

gibt an, wieviel Speicher noch mit der Verwaltung durch diesen Header erreichbar ist.

Wie schon gesagt wurde, ist dem System nicht bekannt, welcher Speicher belegt, sondern nur, welcher Speicher unbelegt ist. Die unbelegten Speicherbereiche sind mit Hilfe einer MemChunk-Struktur miteinander verknüpft. Mit diesen Strukturen lassen sich problemlos die Größe und Position der freien Speicherblöcke bestimmen. Die Struktur hat folgendes Aussehen:

```
struct MemChunk {  
    0 struct MemChunk *mc_Next;  
    4 ULONG mc_Bytes;  
};
```

**mc_Next*

ist ein Zeiger auf die nächste MemChunk-Struktur.

mc_Bytes

gibt an, wieviel Bytes in diesem Speicherblock frei sind.

Der *mh_First*-Eintrag in der MemHeader-Struktur zeigt auf die erste MemChunk-Struktur, die am Anfang des ersten freien Speicherblocks steht. Die ersten vier Bytes dieses freien Blocks sind somit der Zeiger auf den nächsten freien Speicherbereich (auf die nächste MemChunk-Struktur). Die nächsten vier Bytes geben an, wie groß der Speicherbereich ist. In der letzten

MemChunk-Struktur ist der `mc_Next`-Zeiger auf Null gestellt und `mc_Bytes` gibt somit an, wieviel Bytes noch zwischen der jetzigen Speicherposition und dem Wert, der in `mh_Upper` vermerkt ist, liegen.

Eine MemHeader-Struktur verwaltet das gesamte Chip-Memory und eine weitere das Fast-Memory, sofern eines vorhanden ist. Diese Strukturen sind einer Liste verkettet, die in der ExecBase-Struktur enthalten ist. Die Liste hat den Namen 'MemList' und befindet sich bei Offset 322.

Die MemHeader-Priorität für den Fast-Memory-Bereich ist Null, und die Priorität für den Chip-Memory-Bereich ist -10. Deswegen wird auch versucht, immer zuerst Fast-Memory zu belegen und danach erst Chip-Memory.

Soll jetzt Speicher belegt werden, so wird in der MemListe der ExecBase-Struktur nachgesehen, ob die angegebenen Bedingungen in der `AllocMem()`-Funktion auch den Bedingungen der MemHeader-Struktur entsprechen. Als zweites wird geprüft, ob der freie Speicher, der in `mh_Free` vermerkt ist, ausreicht, damit die angegebene Speichermenge reserviert werden kann. Sollte eines der beiden Bedingungen nicht erfüllt sein, so wird nachgesehen, ob noch eine weitere MemHead-Struktur vorhanden ist. Wenn keine weitere erreichbar sein sollte, wird eine negative Rückmeldung von der `AllocMem()`-Funktion übergeben. Ansonsten wird der Zeiger `mh_First` geholt und nachgesehen, ob der in der ersten MemChunk-Struktur angegebene Speicher ausreicht. Wenn nicht, wird zur nächsten MemChunk-Struktur (zum nächsten freien Speicherblock) verzweigt und wiederum verglichen. Wenn kein genügend großer, zusammenhängender Speicher erreichbar ist, wird eine negative Rückmeldung übergeben. Wird hingegen ein ausreichend großer Speicherbereich gefunden, wird errechnet, wieviel von dem freien Speicherblock übrig bleibt, und an der Position, an der der freie Speicher wieder beginnt, eine MemChunk-Struktur eingefügt und diese entsprechend verkettet. Der neubelegte Bereich wird aus der Verkettung entfernt und die Anzahl der belegten Bytes von der Gesamtanzahl abgezogen.

2.5.5 Die Allocate- und Deallocate-Funktion

Es besteht die Möglichkeit, sich eine eigene MemHeader-Struktur zu erstellen und einen separaten Speicherbereich selbständig mit den Funktionen Allocate() und Deallocate() zu verwalten. Mit den Funktionen kann man jedoch nur Speicher belegen und wieder freigeben und hat nicht die Möglichkeit, irgendwelche Bedingungen anzugeben.

Allocate

Speicher = Allocate (MemHeader,ByteSize)
 D0 A0 D0

Offset: -186

Beschreibung: Die Funktion belegt den angegebenen Speicher, der von der angegebenen MemHeader-Struktur verwaltet wird.

Parameter:

MemHeader ist ein Zeiger auf eine MemHeader-Struktur.

ByteSize gibt an wieviel Speicher belegt werden soll.

Speicher ist ein Zeiger auf den belegten Speicher. Sollte kein Speicher gefunden werden, wird eine Null zurückgegeben.

Deallocate

Deallocate (MemHeader,Speicher,ByteSize);
 A0 A1 D0

Offset: -192

Beschreibung: Die Funktion gibt die belegten Speicher wieder an die MemHeader-Struktur zurück.

MemHeader ist ein Zeiger auf die MemHeader-Struktur.

Speicher ist ein Zeiger auf den Anfang des freizugebenden Speichers.

ByteSize gibt an, wieviel Speicher freigegeben werden soll.

Zur Verdeutlichung der Verwendung der Befehle folgt nun ein entsprechendes Programm.

```
#include <exec/execbase.h>
#include <exec/memory.h>
#include <exec/types.h>

#define Byte_Size 10000

struct ExecBase *SysBase;

main()
{
    struct MemHeader *header;
    struct MemChunk *chunk;
    APTR Speicher, AllocMem(), Allocate();

    header = (struct MemHeader *) AllocMem(sizeof(struct MemHeader)
        ,MEMF_PUBLIC | MEMF_CLEAR);
    if (Speicher == 0) {
        printf ("\n AlloMem 1 nicht erfolgreich\n");
        exit (0);
    }

    Speicher = AllocMem(Byte_Size,MEMF_PUBLIC | MEMF_CLEAR);
    if (Speicher == 0) {
        printf ("\n AlloMem 2 nicht erfolgreich\n");
        FreeMem(header,sizeof(struct MemHeader));
        exit (0);
    }

    chunk = (struct MemChunk *) Speicher;
    chunk ->mc_Next = 0;
    chunk ->mc_Bytes = Byte_Size;

    header ->mh_Node.ln_Type = NT_MEMORY;
    header ->mh_Node.ln_Pri = -100;
    header ->mh_Node.ln_Name = "MemHeader";
    header ->mh_Attributes = MEMF_PUBLIC | MEMF_CHIP;
```

```

header ->mh_First = chunk;
header ->mh_Lower = Speicher;
header ->mh_Upper = Speicher + Byte_Size;
header ->mh_Free = Byte_Size;

AddTail (&SysBase ->MemList,header);

Speicher = Allocate (header,500);
if (Speicher == 0) {
    printf ("\n Allocate nicht erfolgreich\n");
    exit (0);
}

Deallocate (header,Speicher,500);
}

```

2.5.6 Beschreibung der restlichen Funktionen

AvailMem

AvailMem (Bedingungen)

D1

Offset: -216

Beschreibung: Die Funktion gibt die Größe des Speichers bezüglich der Bedingungen an. z.B. MEMF_CHIP

AllocAbs

Speicher = AllocAbs (ByteSize,Position)

D0

D0

A1

Offset: -204

Beschreibung: Die Funktion ermöglicht die Belegung eines bestimmten Speicherbereichs, der nicht von Exec gesucht wird, sondern vom Programmierer angegeben werden kann.

Parameter:

ByteSize	gibt die Größe des zu belegenden Speichers an.
Position	ist ein Zeiger auf den zu belegenden Speicher.
Speicher	ist ein Zeiger auf den belegten Speicher, der auch dem angegebenen entspricht. Sollte es nicht möglich sein, den gewünschten Speicher zu belegen, wird eine Null als Fehlermeldung zurückgegeben.

2.6 IO-Handhabung auf dem Amiga

In diesem Kapitel soll weniger auf die Benutzung der verschiedenen Devices eingegangen, als gezeigt werden, wie Exec die IO-Verwaltung vornimmt. Wie man die Ein- und Ausgabe-steuerung des Amiga in seinen eigenen Programmen verwendet, wird im DOS-Teil dieses Buches gezeigt. Die Kenntnis des DOS-Kapitels ist zu empfehlen, um dieses besser verstehen zu können.

2.6.1 Aufbau der IOResult-Struktur

Um Ein- und Ausgabeprozesse erledigen zu können, brauchen wir eine 'IOResult'-Struktur, über die wir unsere Befehle an das Device übergeben können.

Es gibt zwei Arten von 'IOResult'-Strukturen, die sich 'IOResult' und 'IOStdReq' (IO-Standard-Request) nennen. Die 'IOStdReq'-Struktur ist eine Erweiterung der 'IOResult'-Struktur. Die Strukturen haben folgendes Aussehen:

```
struct IOResult {  
0   struct Message io_Message;  
20   struct Device *io_Device;  
24   struct Unit *io_Unit;  
28   UWORD io_Command;  
30   UBYTE io_Flags;  
31   BYTE io_Error;  
};
```

io_Message

ist eine 'Message'-Struktur, wie sie in Kapitel 2.4 beschrieben wurde. Sie wird gebraucht, damit das Device uns mitteilen kann, daß es mit der Bearbeitung des IO-Commands fertig ist. Die 'Message'-Struktur muß korrekt initialisiert werden, bevor die Ein- und Ausgabe funktionieren kann.

**io_Device*

ist ein Zeiger auf die zu benutzende Device-Struktur, die noch beschrieben wird.

**io_Unit*

ist ein Zeiger auf eine Unit-Struktur, deren Beschreibung ebenfalls folgt.

io_Command

ist ein Wort, in dem der auszuführende Befehl übergeben wird.

io_Flags

wird benötigt, um Device-spezifische Statusmeldungen oder Befehle übergeben zu können. Das Byte ist in High- und Low-Nibbel unterteilt. Die unteren vier Bits werden von Exec für interne Zwecke benutzt. Die oberen vier Bits können vom Programmierer benutzt werden, um mit dem Device zu kommunizieren.

io_Error

wird benötigt, um Fehlermeldungen an den Programmierer zu übergeben.

Oft reicht diese Struktur nicht aus, um ein Device benutzen zu können. Für den Fall existiert noch eine weitere Struktur, die dem Benutzer mehr Möglichkeiten bietet. Sie sieht wie folgt aus:


```
    struct IOStdReq {  
0      struct Message io_Message;  
20     struct Device *io_Device;  
24     struct Unit   *io_Unit;  
28     UWORD io_Command;  
30     UBYTE io_Flags;  
31     BYTE io_Error;  
32     ULONG io_Actual;  
36     ULONG io_Length;  
40     APTR io_Data;  
44     ULONG io_Offset;  
    };
```

io_Actual

gibt die Anzahl der zu übertragenden Bytes an. Der Wert kann erst nach der Beendigung der Übertragung ausgelesen werden.

io_Length

gibt die Anzahl der zu übertragenden Bytes an. Dieser Wert muß vor der Übertragung initialisiert werden. Oft wird der Wert auf -1 gesetzt, um eine variable Anzahl von Bytes zu übertragen.

io_Data

ist ein Zeiger auf den Datenpuffer, in den die Daten übertragen werden sollen.

io_Offset

gibt den Offset an, der Device-spezifisch benutzt wird. Beim Trackdisk-Device wird im Offset der Block angegeben, der benutzt werden soll.

2.6.2 Aufbau eines Devices

Die Device-Struktur hat das Aussehen einer Library:

```

struct Device {
    struct Library dd_Library;
};

#define DEV_BEGINIO (-30L)
#define DEV_ABORTIO (-36L)
#define IOB_QUICK 0L
#define IOF_QUICK (1L<<0)
#define CMD_INVALID 0L
#define CMD_RESET 1L
#define CMD_READ 2L
#define CMD_WRITE 3L
#define CMD_UPDATE 4L
#define CMD_CLEAR 5L
#define CMD_STOP 6L
#define CMD_START 7L
#define CMD_FLUSH 8L
#define CMD_NONSTD 9L

```

Um ein Device benutzen zu können, müssen Sie es zuvor öffnen. Der Befehl zum Öffnen eines Device lautet:

```

Error = OpenDevice (Name,Unit,IOResult,flags)
D0                A0  D0    A1    D1

```

Vor der Verwendung der 'OpenDevice()'-Funktion muß die IOResult-Struktur initialisiert worden sein.

Wie auch die Libraries verfügt jedes Device über eine Sprungtabelle, deren Einsprünge mit negativen Offsets erreicht werden. Die so erreichbaren Funktionen dienen zum Öffnen und Schließen eines Device sowie zum Ausführen eines IOs. Eine solche Routine ist nötig, damit eine Funktion wie beispielsweise 'OpenDevice()' jedes Device öffnen kann, auch wenn die zu erledigenden Tätigkeiten von Device zu Device verschieden sind. In der 'OpenDevice()'-Funktion wird dann für die devicespezifischen Prozesse in die Routine des entsprechenden Device eingegriffen.

Die wichtigsten Funktionen, die jedes Device zur Verfügung stellt, sind:

Offset	Funktion
-36	AbortIO
-30	BeginnIO (IO-Ausführen)
-12	Close
-6	Open

Sehen wir uns doch anhand der Assembler-Routine einmal genauer an, was geschieht, wenn ein Device geöffnet werden soll.

Die hier gezeigte Routine ist der wichtigste Teil der 'OpenDevice()' -Funktion, wird jedoch nicht von der Exec-Library direkt, sondern letztendlich von einer Routine über die RAM-Library aufgerufen.

In die Routine wird eingesprungen mit:

D0 = Unit
 D1 = Flags
 A0 = Zeiger auf Devicename
 A1 = Zeiger auf IORequest
 A6 = Zeiger auf ExecBase

```
fc0666 move.l A2,-(A7)    A2 retten
fc0668 move.l A1,A2       Zeiger auf IORequest nach A2
fc066a clr.b 31(A1)       Error-Flag löschen
fc066e movem.l D1-D0,-(A7) D0 und D1 retten
fc0672 move.l A0,A1       Zeiger auf Name nach A1
fc0674 lea 350(A6),A0      Zeiger auf DeviceList nach A0
fc0678 addq.b #1,295(A6)   Forbit
fc067c bsr.l $fc165a       Namen in DeviceList suchen
                           (FindName())
fc0680 move.l D0,A0        Zeiger auf Device nach A0
fc0682 movem.l (A7)+,D1-D0 D0 und D1 zurückholen
fc0686 move.l A0,20(A2)    Zeiger auf Device in IORequest
                           eintragen
fc068a beq.s $fc06ac       Fehler, Device nicht vorhanden
fc068c clr.l 24(A2)        Zeiger auf Unit löschen
```

fc0690	move.l	A2,A1	Zeiger auf IORequest nach A1
fc0692	move.l	A6,-(A7)	A6 retten
fc0694	move.l	A0,A6	Zeiger auf Device nach A6
fc0696	jsr	-6(A6)	spring in OpenDevice ein
fc069a	move.l	(A7)+,A6	A6 zurückholen
fc069c	move.b	31(A2),D0	Error-Flag nach D0
fc06a0	ext.w	D0	Fehler vorzeichenerweitern
fc06a2	ext.l	D0	Fehler vorzeichenerweitern
fc06a4	jsr	-138(A6)	Permit()
fc06a8	move.l	(A7)+,A2	A2 zurückholen
fc06aa	rts		Rücksprung

Einsprung bei Device nicht gefunden (Fehler):

fc06ac	moveq	#\$ff,D0	Fehlerwert nach D0
fc06ae	move.b	D0,31(A2)	in Error-Flag schreiben
fc06b2	bra.s	\$fc06a4	unbedingter Sprung

Auch die nun folgende Routine ist nur der wichtigste Auszug aus der 'CloseDevice()' -Funktion.

Die Routine wird angesprungen mit:

A1 = Zeiger auf IORequest
A6 = Zeiger auf ExecBase

fc06b4	addq.b	#1,295(A6)	Forbit
fc06b8	move.l	A6,-(A7)	A6 retten
fc06ba	move.l	20(A1),A6	Zeiger auf Device nach A6
fc06be	jsr	-12(A6)	einspringen in CloseDevice
fc06c2	move.l	(A7)+,A6	A6 zurückholen
fc06c4	jsr	-138(A6)	Permit
fc06c8	rts		Rücksprung

Als Beispiel für die Routine OpenDevice, die über Offset -6 vom Device ausgehend aufgerufen wird und Device-spezifische Aufgaben beim Öffnen des Device übernimmt, wird jetzt die OpenDevice-Routine des Trackdisk-Device näher erläutert.

Die Unitnummer gibt beim Trackdisk-Device die Laufwerknummer des Laufwerks an, das angesprochen werden soll. Für jedes der vier möglichen Laufwerke ist ein Zeiger in der Device-Struktur reserviert, der, sofern das Laufwerk vorhanden

ist, auf einen entsprechenden Message-Port für das Laufwerk zeigt. Dieser Port, wie auch das Device selbst, hat zusätzlich zu den standardmäßigen Einträgen, die in der C-Struktur festgelegt sind, noch eigene, nicht standardisierte Einträge, wie beispielsweise einen Zähler für die Anzahl der Zugriffe in der Message-Port-Struktur.

Die Routine wird aufgerufen mit:

D0 = Unitnummer

D1 = Flags

A1 = Zeiger auf IORequest

A6 = Zeiger auf Device

```

fe9f42 movem.l A4/A2/D2, -(A7)    D2, A2, A4 retten
fe9f46 move.l  A1, A4             Zeiger auf IORequest nach A4
fe9f48 move.l  D0, D2             Laufwerknummer nach D2
fe9f4a cmpi.l  #$00000004, D0     Nummer zu groß?
fe9f50 bcs.s   $fe9f56            verzweige, wenn Nummer ok
fe9f52 moveq   #$20, D0           sonst Fehlernummer nach D0
fe9f54 bra.s   $fe9f82            Fehler Übergeben, Ende
fe9f56 lsl.w   #2, D0             Nummer *4 für Offset
fe9f58 lea     36(A6), A2         Zeiger auf Laufwerksport
fe9f5c adda.l  D0, A2             Offset hinzaddieren
fe9f5e move.l  (A2), A0           Laufwerksport nach A0
fe9f60 move.l  A0, D0             ist Laufwerk vorhanden?
fe9f62 bne.s   $fe9f70            verzweige, wenn alles ok
fe9f64 bsr.l   $fe9d3e            sonst Laufwerksport bestimmen
fe9f68 tst.l   D0                Port gefunden?
fe9f6a bne.l   $fe9f82            verzweige, wenn nicht gefunden
fe9f6e move.l  A0, (A2)           Port in Device eintragen
fe9f70 move.l  A0, 24(A4)         Port in IORequest eintragen
fe9f74 addq.w  #1, 32(A6)         Anzahl der Zugriffe beim Device
                                erhöhen
fe9f78 addq.w  #1, 36(A0)         Anzahl der Zugriffe beim
                                Laufwerksport erhöhen
fe9f7c movem.l (A7)+, A4/A2/D2    D2, A2, A4 zurückholen
fe9f80 rts                                Rücksprung

```

Einsprung bei Fehler bei der Laufwerksportzuweisung:

```

fe9f82 move.b  D0, 31(A4)         Fehlernummer ins Error-Flag
fe9f86 moveq   #$ff, D0          Zeichen für Fehler nach D0
fe9f88 move.l  D0, 24(A4)         Zeiger auf Unit löschen

```

fe9f8c	move.l	D0,20(A4)	Zeiger auf Device löschen
fe9f90	bra.s	\$fe9f7c	unbedingter Sprung

Die Funktion 'CloseDevice()' springt mit dem Offset -12 vom Device aus in eine Device-eigene Routine ein. Die Routine hat für das Trackdisk-Device folgendes Aussehen:

fe9f92	movem.l	A3-A2, -(A7)	A2 und A3 retten
fe9f96	move.l	A1,A2	Zeiger auf IORequest nach A2
fe9f98	move.l	24(A2),A3	Zeiger auf Laufwerksport nach A3
fe9f9c	subq.w	#1,36(A3)	Anzahl der Zugriffe verringern
fe9fa0	bne.s	\$fe9fa8	verzweige, wenn Laufwerk noch benötigt wird
fe9fa2	bset	#3,64(A3)	Flag setzten
fe9fa8	subq.w	#1,32(A6)	Anzahl der Zugriffe auf das Device verringern
fe9fac	moveq	#\$ff,D0	Löschwert laden
fe9fae	move.l	D0,24(A2)	Zeiger auf Port löschen
fe9fb2	move.l	D0,20(A2)	Zeiger auf Device löschen
fe9fb6	movem.l	(A7)+,A3-A2	A2 und A3 wieder herstellen
fe9fba	moveq	#\$00,D0	Rückmeldung Null nach D0
fe9fbc	rts		Rücksprung

2.6.3 IO-Steuerung über Funktionen

Zu einem Device gehört immer auch ein Task, dem die Befehle übergeben werden. Um einem Device einen Befehl übergeben zu können, gibt es die folgenden Funktionen:

DoIO

Fehler = DoIO(IORequest)
 D0 A1

Offset: -456

Beschreibung: Diese Funktion wird meistens für die Ein-/Ausgabesteuerung verwendet. Sie wartet, bis der übergebene Befehl beendet wird, und kehrt erst dann wieder zum eigentlichen Programm

zurück. Während des Wartens wird der Task auf 'Wait' gesetzt.

SendIO

```
SendIO(IORequest)  
      A1
```

Offset: -462

Beschreibung: Die Funktion dient zum Senden eines IOs an das entsprechende Device, wartet jedoch nicht auf dessen Beendigung.

CheckIO

```
fertig = CheckIO(IORequest)  
      D0      A1
```

Offset: -468

Beschreibung: Die Funktion prüft, ob ein bestimmter IO-Prozess bereits abgearbeitet wurde. Sollte dies der Fall sein, so wird in D0 der Zeiger auf die entsprechende 'IORequest'-Struktur zurückgegeben. Ist der IO-Prozess nicht fertig, wird eine Null übergeben.

WaitIO

```
WaitIO(IORequest)  
      A1
```

Offset: -474

Beschreibung: Die Funktion wartet so lange, bis der IO-Prozess erledigt ist. Während dieser Zeit wird der laufende Task auf Wait gesetzt, um andere Tasks abarbeiten zu können. Die Funktionen 'SendIO' und 'WaitIO' ergeben zusammengesetzt den Befehl 'DoIO'.

AbortIO

AbortIO(IORequest)

A1

Offset: -480

Beschreibung: Die Funktion beendet einen IO-Prozess, wie sich auch schon anhand des Namens ansehen läßt.

Nach dieser Kurzbeschreibung der Funktionen wollen wir uns einmal ansehen, wie diese Funktionen im Betriebssystem aussehen.

Die 'DoIO'-Assemblerroutine hat folgendes Aussehen:

In A1 steht ein Zeiger auf die zuvor errichtete IORequest-Struktur.

fc06dc	move.l	A1,-(A7)	A1 retten
fc06de	move.b	#01,30(A1)	Quick-Bit setzten
fc06e4	move.l	A6,-(A7)	A6 retten
fc06e6	move.l	20(A1),A6	Zeiger auf Device holen
fc06ea	jsr	-30(A6)	zu IO-Ausführen springen
fc06ee	move.l	(A7)+,A6	A6 holen
fc06f0	move.l	(A7)+,A1	A1 holen

Ab hier beginnt die Funktion 'WaitIO', die von der 'DoIO'-Funktion benutzt wird.

fc06f2	btst	#0,30(A1)	Quick-Bit testen
fc06f8	bne.s	\$fc0744	fertig, wenn gesetzt
fc06fa	move.l	A2,-(A7)	A2 retten
fc06fc	move.l	A1,A2	Zeiger auf IORequest nach A2
fc06fe	move.l	14(A2),A0	Zeiger auf Reply-Port
fc0702	move.b	15(A0),D1	Signalbit für Port holen
fc0706	moveq	#00,D0	D0 löschen
fc0708	bset	D1,D0	Bit für Signal setzen
fc070a	move.w	#\$4000,\$dff09a	Disable-
fc0712	addq.b	#1,294(A6)	Macro
fc0716	cmpi.b	#\$07,8(A2)	Type der Msg = Reply MSG?
fc071c	beq.s	\$fc0724	verzweige, wenn Type ok
fc071e	jsr	-318(A6)	sonst auf Msg warten (Wait())
fc0722	bra.s	\$fc0716	unbestimmter Sprung

fc0724	move.l	A2,A1	IORequest nach A1
fc0726	move.l	(A1),A0	
fc0728	move.l	4(A1),A1	Node aus Reply-Msg.-liste
fc072c	move.l	A0,(A1)	entfernen
fc072e	move.l	A1,4(A0)	
fc0732	subq.b	#1,294(A6)	Enable-
fc0736	bge.s	\$fc0740	
fc0738	move.w	#\$c000,\$dff09a	Macro
fc0740	move.l	A2,A1	Zeiger auf IORequest nach A1
fc0742	move.l	(A7)+,A2	A2 herstellen
fc0744	move.b	31(A1),D0	Fehler-Flag nach D0
fc0748	ext.w	D0	vorzeichenerweitern
fc074a	ext.l	D0	vorzeichenerweitern
fc074c	rts		Rücksprung

In der obigen Routine wird als erstes das Quick-Bit gesetzt. Dann wird der Zeiger auf das Device nach A6 gebracht und in die BeginIO-Funktion eingesprungen, die später noch anhand des Trackdisk-Device beschrieben wird. In dieser Routine wird der auszuführende Befehl auf seine Gültigkeit überprüft und gegebenenfalls an den Trackdisk-Task übergeben. Kehrt das Programm aus dieser Routine zurück, ist der Message-Type in der 'IORequest'-Struktur immer auf 'Message' gestellt. Dem Task wird in der Routine die 'IORequest'-Struktur als Message übergeben.

An dieser Stelle ist die Befehlsübergabe schon abgeschlossen. Es wird jetzt lediglich auf deren Beendigung gewartet. Sollte das Quick-Bit nicht gelöscht worden sein, ist die Routine jetzt zu Ende. Ansonsten muß geprüft werden, ob der IO-Prozess beendet wurde. Für diese Überprüfung reicht es aus, den Typ der 'Message'-Struktur auf 'Reply-Msg' zu testen. Ist das nicht der Fall, geht der Task in Wartestellung, bis eine entsprechende Message eingetroffen ist.

Es ist wahrscheinlich nötig, genauer zu erklären, warum es ausreicht, die Unterstruktur 'Message' in der 'IORequest'-Struktur auf den Type Reply-Msg. zu prüfen.

Von der 'BeginIO'-Routine (die Routine, die vom Device zur Verfügung gestellt wird (Offset -30)) wird eine Message zu dem entsprechenden Task, der die Befehlsbearbeitung übernimmt,

geschickt. Als Message wird hier unsere 'IORequest'-Struktur gesandt, was mit der Funktion 'PutMsg()' geschieht. In der Funktion wird der Type der zu schickenden Message automatisch auf 'Message' gestellt (Bytewert 05). Unsere 'IORequest'-Struktur hat zwar immer noch die gleiche Position im Speicher, ist jetzt jedoch mit ihrer 'Node'-Struktur in die Message-Liste des Tasks eingehängt. Der Task arbeitet unseren Befehl ab und sendet eine Reply-Message als Rückmeldung, daß er mit seiner Arbeit fertig ist. Als Message wird wieder die 'IORequest'-Struktur gesandt. Von der 'ReplyMsg()'-Funktion wird der Type der zu sendenden Message wieder automatisch auf ReplyMsg (Bytewert 07) gestellt und in die Message-Liste des Reply-Ports eingehängt. Die 'WaitIO()'-Funktion prüft den Type der Message-Struktur in unserer 'IORequest'-Struktur, stellt fest, daß es sich um eine Reply-Message handelt (sie wurde ja soeben als solche gesandt) und muß nun noch die Reply-Message, die in die ReplyMsg-Liste eingefügt wurde, wieder entfernen.

Beschreibung der Funktion SendIO:

In A1 steht der Zeiger auf die 'IORequest'-Struktur.

fc06ca clr.b	30(A1)	alle Flags löschen
fc06ce move.l	A6,-(A7)	A6 retten
fc06d0 move.l	20(A1),A6	Zeiger auf Device holen
fc06d4 jsr	-30(A6)	zu BeginIO springen
fc06d8 move.l	(A7)+,A6	A6 zurückholen
fc06da rts		Rücksprung

Sie sehen, daß die 'SendIO'-Funktion nicht anders ist als der erste Teil der 'DoIO'-Funktion.

Beschreibung der Funktion CheckIO:

In A1 steht der Zeiger auf die 'IORequest'-Struktur.

fc074e btst	#0,30(A1)	prüfe ob Quick-Bit gesetzt
fc0754 beq.s	\$fc075a	verzweige, wenn nicht gesetzt
fc0756 move.l	A1,D0	ansonsten ok Meldung übernehmen
fc0758 rts		Rücksprung
fc075a cmpi.b	#\$07,8(A1)	Type der Message-Struktur = Replymsg?

fc0760 beq.s	\$fc0766	ja, dann positive Rückmeldung
fc0762 moveq	#\$00,D0	ansonsten negative Rückmeldung
fc0764 rts		Rücksprung
fc0766 move.l	A1,D0	ok Meldung übergeben
fc0768 rts		Rücksprung

Die 'CheckIO'-Funktion prüft lediglich, ob eine Reply-Message eingegangen ist. Sollte das der Fall sein, wird der Zeiger auf die 'IORequest'-Struktur in D0 übergeben, sonst wird einen Null in D0 zurückgesand.

Sie sehen, daß in dieser Routine zwar geprüft wird, ob eine Reply-Message angekommen ist, diese jedoch nicht aus der Liste der Reply-Messages entfernt wird. Das Entfernen der Reply-Message muß bei der Benutzung der 'CheckIO'-Funktion von Hand erledigt werden, z.B. mit der 'GetMsg()'-Funktion, sofern das Quick-Bit nicht gesetzt ist.

Beschreibung der Funktion AbortIO:

In A1 steht der Zeiger auf die 'IORequest'-Struktur.

fc076a move.l	A6,-(A7)	A6 retten
fc076c move.l	20(A1),A6	Zeiger auf Device nach A6
fc0770 jsr	-36(A6)	In AbortIO einspringen
fc0774 move.l	(A7)+,A6	A6 zurückholen
fc0776 rts		Rücksprung

Wie schon gesagt, verfügt jedes Device über eine kleine Sprungtabelle für die Verwaltung des Device. Sehen wir uns anhand des Trackdisk-Device die Routine, die sich hinter IO-Ausführen verbirgt, einmal näher an. Sie wird mit Offset -30 aufgerufen und von den Funktionen 'DoIO' und 'SendIO' verwendet.

In A1 steht der Zeiger auf die 'IORequest'-Struktur und in A6 der Zeiger auf das Device.

fe9fbe clr.b	31(A1)	Errorflag löschen
fe9fc2 moveq	#\$00,D0	D0 löschen
fe9fc4 move.b	29(A1),D0	io_Command nach D0
fe9fc8 cmpi.b	#\$16,D0	erlaubter Befehl?

fe9fcc bcc.s	\$fea016	verzweige, wenn nicht erlaubt
fe9fce move.l	24(A1),A0	Zeiger auf Device-Port
fe9fd2 move.l	#\$000c61c2,D1	Befehlsentschlüsselbits
fe9fd8 btst	D0,D1	Befehl direkt ausführen?
fe9fda bne.s	\$fe9ff0	ja, Befehl ausführen
fe9fdc andi.b	#\$7e,30(A1)	Flags bis auf Quick-Bit löschen
fe9fe2 move.l	A6,-(A7)	A6 retten
fe9fe4 move.l	52(A6),A6	ExecBase holen
fe9fe8 jsr	-366(A6)	PutMsg (IORequest an Trackdisk- Task übergeben)
fe9fec move.l	(A7)+,A6	A6 holen
fe9fee bra.s	\$fea014	unbedingter Sprung
fe9ff0 bset	#7,30(A1)	Flag für Ausführen setzen
fe9ff6 move.b	#\$05,8(A1)	Type in 'IORequest'-Struktur auf 'Message', damit WaitIO warten muß
fe9ffc movem.l	A3-A2,-(A7)	A2 und A3 retten
fea000 move.l	A0,A3	Zeiger auf Drive-Port nach A3
fea002 move.l	A1,A2	Zeiger auf IORequest nach A2
fea004 lea	762(PC)(=\$fea300),A0	Zeiger auf Befehlstab.
fea008 lsl.w	#2,D0	Befehl *4, um Offset zu bekommen
fea00a move.l	0(A0,D0.W),A0	Einsprung holen
fea00e jsr	(A0)	Einspringen
fea010 movem.l	(A7)+,A3-A2	A2 und A3 wieder holen
fea014 rts		Rücksprung

2.7 Interrupt-Handhabung auf dem Amiga

In diesem Kapitel wollen wir uns einmal ansehen, wie der Amiga seine sieben vom Prozessor zur Verfügung gestellten Interrupt-Ebenen nutzt, und wie man sie für sich nutzen kann.

Zum besseren Verständnis des Kapitels sollten Sie den Interrupt-Teil aus Kapitel 1 dieses Buches schon gelesen haben.

In diesem Kapitel wird der Interrupt erst betrachtet, nachdem der Prozessor ein entsprechendes Signal von der 4703 Interrupt Logic bekommen hat.

Nachdem der Prozessor ein Interrupt-Signal erhalten hat und dieses nicht gesperrt war, lädt er die zu der entsprechenden

Priorität gehörige Adresse in seinen Programmzeiger und führt den Interrupt an entsprechender Stelle weiter. Die Vektoren für die Fortsetzung des Interrupts stehen ab Adresse \$0064 bis \$007F, wobei die ersten vier Bytes den Vektor für Interrupt-Ebene 1 und die Bytes von \$007C bis \$007F den Vektor für Interrupt-Ebene 7 beinhalten.

Da beim Amiga mehr Interrupts gebraucht werden, als der Prozessor zur Verfügung stellt, laufen alle Interrupts erst über ein Register, mit dem 15 verschiedene Interrupts verwaltet werden. Die eingegangenen Interrupts werden mit Hilfe eines Interrupt-Enable-Registers auf ihre Zulässigkeit überprüft, worauf dann, sofern der Interrupt erlaubt war, ein Interrupt am Prozessor durchgeführt wird. Da für 15 Interrupts lediglich sieben Prioritäten zur Verfügung stehen, sind mehreren Interrupts gleiche Prozessor-Prioritäten zugeordnet. Aufgrund der gleichen Prioritäten haben somit verschiedene Interrupts einen gleichen Einsprung, wo sie softwaremäßig nach ihren Ursachen entsprechend "sortiert" werden. Die folgende Tabelle zeigt Ihnen, welche Interrupts welche Prioritäten haben. Mit Pseudo-Priorität ist die Priorität gemeint, die softwaremäßig abgefragt wird. Ihre Zahl entspricht der Bitnummer des im Interrupt-Request-Register vermerkten Interrupts.

Ein Beispiel zur Verdeutlichung:

Der Interrupt, der ausgelöst wird, wenn der Rasterstrahl des Bildschirms Zeile 0 durchläuft, hat die gleiche Prozessorpriorität wie der Interrupt für das Beenden der Blitteraktivität. Die beiden Interrupts werden an Bit 5 und 6 des Interrupt-Request-Registers angezeigt. Softwaremäßig wird der Interrupt mit der höheren Bitnummer (der höheren Pseudo-Priorität) vor dem mit der niedrigeren abgefragt. In diesem Fall also der Blitter- vor dem Rasterstrahlinterrupt.

Die nächste Tabelle zeigt Ihnen die 15 Interrupts des Amiga mit deren Prozessor- sowie Pseudoprioritäten.

Pseudo-Priorität	Name	Prozessor-Priorität	Funktion
14	INTEN	(6)	Interrupts erlauben
13	EXTER	6	Interrupt von CIA-B oder Expansion-Port
12	DSKSYN	5	Disk-Synchronisationswert erkannt
11	RBF	5	Eingabepuffer des Seriellen Ports voll
10	AUD3	4	Audiodaten Kanal 3 ausgegeben
9	AUD2	4	Audiodaten Kanal 2 ausgegeben
8	AUD1	4	Audiodaten Kanal 1 ausgegeben
7	AUD0	4	Audiodaten Kanal 0 ausgegeben
6	BLIT	3	Blitter fertig
5	VERTB	3	Beginn der vertikalen Austast-lücke erreicht
4	COPER	3	Reserviert für Copper-Interrupts
3	PORTS	2	Interrupt von CIA-A oder Expansion-Port
2	SOFT	1	Reserviert für Software-Interrupts
1	DSKBLK	1	Disk-DMA-Transfer beendet
0	TBE	1	Ausgabepuffer des seriellen Ports leer

2.7.1 Aufbau der Interrupt-Strukturen

Für die Interrupts des Amiga gibt es, was Sie nicht überraschen wird, auch wieder eine Struktur, mit der sie sich gut verwalten lassen.

Die Struktur hat folgendes Aussehen:

```

struct Interrupt {
0   struct Node is_Node;
14  APTR is_Data;
18  VOID (*is_Code)();
};

```

is_Node

ist eine Nodestruktur, wie Sie sie schon des öfteren kennengelernt haben.

is_Data

ist ein Zeiger auf einen Datenspeicher, der vom Interrupt beliebig genutzt werden kann. Die Größe des Datenspeichers kann bei der Erstellung der Struktur beliebig groß gewählt werden.

*is>(*Code)()*

ist ein Zeiger auf das eigentliche Interrupt-Programm, das ausgeführt werden soll.

Es gibt zwei verschiedene Möglichkeiten, einen Interrupt zu nutzen. Zum einen kann man mit einem Interrupt ein Interrupt-Programm ansprechen, was über einen Interrupt-Handler geschieht, oder man kann, mit einem Interrupt mehrere verschiedene Programme aufrufen lassen, was durch Interrupt-Server erledigt wird.

Jeder zugelassene Interrupt ist in der 'ExecBase'-Struktur (die Hauptstruktur von Exec, auf die wir noch zu sprechen kommen) vermerkt. In dieser Struktur befindet sich für jeden der 15 möglichen Interrupts noch eine kleine Unterstruktur, deren richtige Initialisierung für den Interrupt sehr wichtig ist. Die Unterstruktur heißt 'IntVector' und hat folgendes Aussehen:

```
struct IntVector {  
0  APTR iv_Data;  
4  VOID (*iv_Code)();  
8  struct Node *iv_Node;  
};
```

Je nachdem, ob der entsprechende Interrupt von einem Interrupt-Handler oder Server verwaltet wird, ist die Initialisierung der 'IntVector'-Struktur unterschiedlich. Für den Interrupt-Handler sieht diese Initialisierung wie folgt aus:

iv_Data

ist ein Zeiger auf den gleichen Datenspeicher, der auch schon in der Struktur 'Interrupt' vorkam.

*(*iv_Code)()*

ist ein Zeiger auf das Interrupt-Programm, das ausgeführt werden soll.

**iv_Node*

ist ein Zeiger auf die Interrupt-Struktur, die zuvor beschrieben wurde.

Für den Interrupt-Server hat sie dieses Aussehen:

iv_Data

ist ein Zeiger auf eine ServerList-Struktur, auf die wir noch zu sprechen kommen.

*(*iv_Code)()*

ist ein Zeiger auf eine Routine, die die Verwaltung von mehreren Interrupt-Programmen übernimmt.

**iv_Node*

ist hier nicht belegt und hat den Wert Null.

Diese Interrupt-Vector-Strukturen brauchen nicht von Hand initialisiert zu werden, sofern man einen Interrupt-Handler verwenden will. Die Initialisierung geschieht beim Aufruf der Exec-Funktion 'SetIntVector'. Zur Benutzung eines Interrupts mit einem Interrupt-Handler muß die Interrupt-Struktur initialisiert und daraufhin die 'SetIntVector'-Funktion aufgerufen werden.

Sehen wir uns einmal an, wie ein Interrupt weiterverarbeitet wird, nachdem er vom Prozessor zur Verarbeitung freigegeben

wurde. Als Beispiel für die Verwaltung eines Interrupts folgt jetzt der Assemblerteil, der einen Level 3 (Priorität 3) Interrupt verwaltet.

Bei \$FC0CD8 ist der Einsprung, wo der Prozessor seine Arbeit nach Erkennen eines Level-3-Interrupts fortsetzt. Dieser Adressenwert stimmt nur mit der Kickstartversion des Amiga 500 oder 2000 überein. Beim Amiga 1000 ist die Routine zwar gleich, aber, abhängig von der Kickstartversion, leicht verschoben.

```

movem.l  A6-A5/A1-A0/D1-D0,-(A7)  Register in den Stapel retten
lea      $dff000,A0                Anfang der Register nach A0
move.l   $0004,A6                  SysBase nach A6
move.w   28(A0),D1                 Interrupt enable Register lesen
btst     #14,D1                    Master-Bit testen
beq.l    L1                         kein Interrupt zugelassen
and.w    30(A0),D1                 mit Interrupt request Register die
                                   zulässigen Interrupts herausfiltern
btst     #6,D1                     Bit für Blitter done testen
beq.s    L2                         nein, anderes Bit testen
movem.l  156(A6),A5/A1             aus IntVector Struktur Zeiger auf
                                   Daten und Programm holen
pea      -36(A6)                   Rücksprung durch RTS Rücksprungadresse auf
                                   ExitInter stellen
jmp      (A5)                      in Interrupt einspringen

```

L2:

```

btst     #5,D1                     Bit für Rasterinter testen
beq.s    L3                         nein, weiter testen
movem.l  144(A6),A5/A1             aus IntVector Struktur Zeiger auf
                                   Daten und Programm holen
pea      -36(A6)                   Rücksprung auf ExitInter
jmp      (A5)                      Einsprung

```

L3:

```

btst     #4,D1                     Bit für Copper Interrupt testen
beq.s    L1                         nein, dann Ende
movem.l  132(A6),A5/A1             Zeiger aus Daten und Programm holen
pea      -36(A6)                   Rücksprung auf ExitInter
jmp      (A5)                      Einsprung

```

L1:

```
movem.l (A7)+,A6-A5/A1-A0/D1-D0  Register wieder  
rte                               herstellen, Rücksprung
```

Anhand des Assemblerlistings können wir erkennen, welche Register wir in einem eigenen Interrupt-Programm gefahrlos verwenden dürfen und welche mit bestimmten Werten an unser Programm übergeben werden.

Die Register D0, D1, A0, A1, A5 und A6 werden vor dem Einsprung in das eigentliche Interrupt-Programm auf der Stapel gerettet. Zusätzlich wird im Stapel noch die Rücksprungadresse vermerkt, so daß das eigene Interrupt-Programm mit RTS abgeschlossen werden muß.

Beschreibung der Register:

- D0 beinhaltet keine sinnvolle Information.
- D1 beinhaltet die Und-Verknüpfung zwischen IntEnaReg und IntReqReg und zeigt damit an, welche Interrupts zur Zeit erlaubt sind und laufen.
- A1 ist ein Zeiger auf den Anfang der Hardware-Register.
- A5 ist ein Zeiger auf den auszuführenden Code.
- A6 ist ein Zeiger auf SysBase.

Keines dieser Register muß vor dem Rücksprung aus dem Interrupt-Programm wieder auf seinen alten Wert gesetzt werden.

Bei der Benutzung eines Interrupt-Handlers wird mit 'jmp (a5)' in das eigentliche Interrupt-Programm eingesprungen. Das Interrupt-Programm muß mit 'RTS' abgeschlossen werden.

Bei der Verwaltung eines Interrupts durch einen Interrupt-Server sind die einzelnen Interrupt-Programme, die beim Auftreten des entsprechenden Interrupts ausgeführt werden sollen, mit ihren Interrupt-Strukturen in einer Liste verkettet. Die

Reihenfolge ihrer Abarbeitung entspricht der eingetragenen Priorität in ihrer 'is_Node'-Struktur.

Die Listenstruktur, in der die Interrupts enthalten sind, hat folgendes Aussehen:

```
struct ServerList {  
    0   struct List sl_List;  
    14  UWORD   sl_IntClr1;  
    16  UWORD   sl_IntSet;  
    18  UWORD   sl_IntClr2;  
    20  UWORD   sl_pad;  
};
```

sl_List

ist eine List-Struktur, wie sie schon des öfteren zu finden war.

sl_IntClr1 und sl_IntClr2

sind Worte, in denen das Bit gesetzt ist, welches im Interrupt-Request-Register für den Interrupt zuständig ist. Das Clr/Set Bit (Bit 15), das bereits in Kapitel 1 des Buches erklärt wurde, ist hier gelöscht. Schreibt man diesen Wert in das Interrupt-Requeset-Register, wird das Interruptbit gelöscht.

sl_IntSet

ist ein Wort mit dem gleichen Inhalt wie 'sl_IntClr', jedoch mit dem Unterschied, daß hier das Clr/Set Bit gesetzt ist.

sl_pad

ist ein Wort, das immer den Wert Null hat.

Diese Struktur ist in keinem Include-File enthalten und muß vor Gebrauch von Hand eingebunden werden.

Da wir jetzt einen Interrupt-Server benutzen wollen, sieht die Initialisierung der InterruptVector-Struktur so aus, daß in

'(*iv_Code)()' ein Zeiger auf eine Routine steht, die eine Interrupt-Liste verwaltet.

Diese Routine wird mit 'jmp (a5)' angesprungen, liegt bei \$FC12FC und hat folgendes Aussehen:

In A1 befindet sich der Zeiger auf die ServerList-Struktur.

```

fc12fc move.w 18(A1),-(A7)  sl_IntClr lesen und speichern
fc1300 move.l A2,-(A7)    A2 retten
fc1302 move.l (A1),A2     Zeiger auf ersten Interrupt
fc1304 move.l (A2),D0     nachsehen, ob Interrupt vorhanden
fc1306 beq.s  $fc1316     verzweige, wenn nicht vorhanden
fc1308 movem.l 14(A2),A5/A1  a1 = is_Data und
                             A5 = (*is_Code)()
fc130e jsr    (A5)        Einsprung ins Interrupt-Programm
fc1310 bne.s  $fc1316     verzweige, wenn Rückmeldung nicht
                             Null
fc1312 move.l (A2),A2     Zeiger auf nächsten Interrupt
                             stelle
fc1314 bra.s  $fc1304     unbedingter Sprung
fc1316 move.l (A7)+,A2    A2 wieder herstellen
fc1318 move.w (A7)+,$dff09c Interruptbit im
                             Interrupt-Request-Register löschen
fc131e rts              Rücksprung

```

Wenn ein Interrupt abgearbeitet ist, wird der nächste sofort aufgerufen, sofern noch einer vorhanden ist und der vorige als Rückmeldung eine Null übergeben hat. Diese Rückmeldung geschieht meistens in D0.

Wie für die Interrupt-Handler stehen auch hier die gleichen Register zur freien Benutzung zur Verfügung. Sie unterscheiden sich jedoch in der Übergabe sinnvoller Werte zur Weiterbenutzung.

Beschreibung der Register:

- D0 ist ein Zeiger auf den nächsten Interrupt.
- D1 unbestimmter Wert.
- A1 ist ein Zeiger auf den Datenspeicher des Interrupts.

A5 ist ein Zeiger auf das eigene Interrupt-Programm.

A6 unbestimmter Wert.

Zum besseren Verständnis der Verwaltung eines Interrupts durch einen Interrupt-Server wird dies anhand des Port-Interrupts, der von CIAA ausgelöst wird und unter anderem die Tastatur abfragt, noch einmal verdeutlicht.

Die Interrupt-Vector-Struktur liegt für diesen Interrupt ab Offset 120 von der ExecBase-Struktur ausgehend und ist wie folgt initialisiert:

iv_Data

zeigt auf die 'ServeList'-Struktur

*(*iv_Code)()*

ist ein Zeiger auf ein Unterprogramm, das die Server-Liste verwaltet. Diese Routine liegt beim Amiga 500 und Amiga 2000 bei \$FC12FC.

**iv_Node*

ist nicht initialisiert, da die Interrupt-Strukturen in der 'ServerList'-Struktur verkettet sind.

Die initialisierte 'ServerList'-Struktur hat folgendes Aussehen:

sl_List

ist initialisiert wie jede Liststruktur. In dieser Liste sind die 'Interrupt'-Strukturen miteinander verkettet.

sl_IntClr und *sl_IntSet*

haben den Wert \$0008, da Bit 3 des Interrupt-Request-Registers für diesen Interrupt zuständig ist.

sl_IntSet

hat den Wert \$8008.

2.7.2 Soft-Interrupts

Wie sich aus dem Namen schon ersehen läßt, geht es hier um Interrupts, die softwaremäßig (von Hand) ausgelöst werden. Diese Interrupts haben eine höhere Priorität als Tasks, jedoch eine niedrigere als Hardware-Interrupts und können verwendet werden, um beliebige asynchrone Prozesse ablaufen zu lassen.

Um einen solchen Interrupt aufzurufen, dient die 'Cause()'-Funktion. Ruft man sie auf, wird der laufende Task unterbrochen und der Interrupt ausgeführt. Wird die 'Cause()'-Funktion von einem Hardware-Interrupt aufgerufen, so wird dieser zuerst beendet, bevor der Soft-Interrupt ausgelöst wird.

Um einen Soft-Interrupt zu erzeugen, muß, wie auch bei allen anderen Interrupts zuvor, die 'Interrupt'-Struktur entsprechend initialisiert werden. Daraufhin kann die 'Cause()'-Funktion aufgerufen werden, wobei der Zeiger auf die Struktur in A1 übergeben wird.

Einem Soft-Interrupt können nur 5 verschiedene Prioritäten zugewiesen werden, und zwar: -32, -16, 0, +16, +32. Das liegt daran, daß in der 'ExecBase'-Struktur für jede Priorität eine 'SoftIntList'-Struktur zur Verfügung steht und nur Platz für 5 Strukturen vorgesehen ist.

Jede der Strukturen hat folgendes Aussehen:

```
struct SoftIntList {  
0  struct List sh_List;  
14  UWORD sh_Pad;  
};
```

sh_List

ist eine gewöhnliche 'List'-Struktur.

sh_Pad

ist ein Wort, das nur dazu gebraucht wird, die Struktur auf Langwortgröße zu halten. Es hat immer den Wert Null.

Fünf dieser Strukturen befinden sich innerhalb der 'ExecBase'-Struktur. Sie liegen hintereinander und beginnen bei Offset 434.

Wie Exec diese Interrupts verwaltet, wird durch die folgenden Assemblerprogramme verdeutlicht.

Als erstes folgt nun die Dokumentation der Routine 'Cause()':

```

fc1320 move.w  #$4000,$dff09a    Alle Interrupts sperren
fc1328 addq.b   #1,294(A6)        IdNestCount hochzählen
fc132c cmpi.b   #$0b,8(A1)        Type auf SoftInt testen
fc1332 beq.s    $fc1370           verzweige, wenn Type SoftInt
fc1334 move.b   #$0b,8(A1)        ansonsten neu
fc133a moveq    #$00,D0           eintragen
fc133c move.b   9(A1),D0          Priorität noch D0
fc1340 andi.w   #$00f0,D0         unerlaubte Bits löschen
fc1344 ext.w    D0                und vorzeichenrichtig erweitern
fc1346 lea      466(A6),A0        Zeiger auf Int. mit Pri. 0
fc134a adda.w   D0,A0            Position der SoftIntList anhand
                                der Priorität bestimmen

fc134c lea      4(A0),A0          Zeiger auf lh_Tail
fc1350 move.l   4(A0),D0          Zeiger auf letztes Glied nach D0
fc1354 move.l   A1,4(A0)         neuen Interrupt als letzten
                                eintragen

fc1358 move.l   A0,(A1)          Int.-Nachfolger auf Null setzen
fc135a move.l   D0,4(A1)         Zeiger auf Vorgänger setzen
fc135e move.l   D0,A0            Zeiger auf früheren Int.
fc1360 move.l   A1,(A0)          dessen Nachfolger auf jetzigen
fc1362 bset     #5,292(A6)        im SysFlag SoftInt erlauben
fc1368 move.w   #$8004,$dff09c   Interrupt verursachen
fc1370 subq.b   #1,294(A6)        IdNestCount verringern
fc1374 bge.s    $fc137e          verzweige, wenn Int. noch nicht
                                erlaubt werden darf

fc1376 move.w   #$c000,$dff9a    Interrupts zulassen
fc137e rts                                Rücksprung

```

Anhand des Assemblerlistings können Sie sehen, wie ein Soft-Interrupt erzeugt wird. Bevor der Interrupt ausgeführt werden kann, muß dieser zuvor in eine der fünf 'SoftIntList'-Strukturen

eingetragen werden. In welche dieser Strukturen er eingetragen wird, hängt von seiner Priorität ab. Aus der Ermittlung der Struktur in die er eingetragen werden soll, läßt sich ersehen, warum nur die Prioritäten -32, -16, 0, +16, +32 zulässig sind. Jede 'SoftIntList'-Struktur ist 16 Bytes lang. Es wird ein Zeiger auf die mittlere Struktur erzeugt. Zu diesem Zeiger wird die Priorität hinzuaddiert, wodurch die Position der gewünschten Struktur ermittelt wird.

Nachdem die zugehörige 'SoftIntList'-Struktur festgestellt wurde, wird die Struktur des zu erzeugenden Interrupts als letztes Glied in die Liste eingehängt und in der 'ExecBase'-Struktur im 'SysFlags' vermerkt, das ein Soft-Interrupt anliegt. Daraufhin wird im Interrupt-Request-Register das Bit zur Ausführung eines Soft-Interrupts gesetzt und dieser auch in der 'ExecBase'-Struktur als erlaubt gekennzeichnet. Nachdem alles erledigt ist, werden die Interrupts, die zu Beginn der Routine gesperrt wurden, wieder freigegeben.

Der Soft-Interrupt wird jetzt ausgeführt und aus der 'InterruptVector'-Struktur der Zeiger auf das auszuführende Programm geholt, welches die 'SoftIntList'-Strukturen verwaltet. Das Programm liegt ab \$FC1380.

```

fc1380 move.w  #$0004,$dff09c  Interrupt-Request-Bit löschen
fc1388 bclr    #5,292(A6)      'SysFlag'-Bit löschen, testen
fc138e bne.s   $fc1392         verzweige, wenn Interrupt
                                erlaubt
fc1390 rts                                           Rücksprung
fc1392 move.w  #$0004,$dff09a  Interrupt-Enable-Bit löschen
fc139a bra.s   $fc13c2         unbedingter Sprung
fc139c move    #$2700,SR       alle Interrupts vom Prozessor
                                erlauben
fc13a0 move.l  (A0),A1         Zeiger auf ersten Interrupt
fc13a2 move.l  (A1),D0         ist Node noch gültig?
fc13a4 beq.s   $fc13ae         verzweige, wenn kein Int. mehr
fc13a6 move.l  D0,(A0)         ersten Interrupt aus Liste
                                löschen
fc13a8 exg     D0,A1           A1 und D1 austauschen
fc13aa move.l  A0,4(A1)        ln_Pred auf Liste stellen
fc13ae move.l  D0,A1           Zeiger auf ersten Interrupt
fc13b0 move.b  #$02,8(A1)     ln_Type auf Int. setzen
fc13b6 move    #$2000,SR       alle Interrupts sperren

```



```

fc13ba movem.l 14(A1),A5/A1  A1 = is_Data, A5 = (*is_Code)()
fc13c0 jsr      (A5)          einspringen
fc13c2 moveq    #$04,D0       Anzahl der Int. Listen -1
fc13c4 lea      498(A6),A0     Zeiger auf Int. Liste mit
                               höchster Priorität
fc13c8 move.w   #$0004,$dff09c Int.-Request-Bit löschen
fc13d0 cmpa.l   8(A0),A0       Liste leer?
fc13d4 bne.s    $fc139c        verzweige, wenn nicht leer
fc13d6 lea      -16(A0),A0     sonst Zeiger auf Int. Liste mit
                               niedrigerer Priorität setzen
fc13da dbf      D0,$fc13d0     verzweige, wenn nicht alle
                               Listen durchsucht
fc13de move     #$2100,SR      alle Interrupts bis auf Pri. 1
                               sperren
fc13e2 move.w   #$8004,$dff09a Soft-Interrupt wieder
                               erlauben
fc13ea rts                               Rücksprung

```

Als erstes wird in der soeben gezeigten Routine geprüft, ob das Bit, das die Zulassung eines Soft-Interrupts angibt, gesetzt ist, wie es von der 'Cause()-Funktion erledigt wird. Sollte dies der Fall sein, wird die Routine ausgeführt. Als nächstes wird das Interrupt-Request-Bit gelöscht und die Soft-Interrupt-Listen auf ihre Interrupt-Strukturen durchsucht. Ist eine Liste abgearbeitet oder leer, wird die nächste Liste durchsucht. Die Listen werden in fallender Priorität bearbeitet.

2.7.3 Die CIA-Interrupts

Nachdem wir die Interruptverwaltung des Amiga besprochen haben, soll auf die Interrupts, die von den CIAs ausgelöst werden, hier gesondert eingegangen werden. Die Prozessor-Prioritäten der beiden Bausteine sind 6 für CIAB und 2 für CIAA und werden beide von einem Interrupt-Server verwaltet. Folglich zeigt das 'iv_Code' Element der 'InterruptVektor'-Struktur auf eine 'ServerList'-Struktur und '(*iv_Code)()' auf die Routine zur Verwaltung der Server-Liste. Die 'InterruptVektor'-Strukturen der Interrupts liegen für CIAA ab Offset 120 und für CIAB ab Offset 240 in der 'ExecBase'-Struktur.

2.7.3.1 Die CIA-Resource-Struktur

Normalerweise befindet sich in der Server-Liste der Interrupts nur eine Interrupt-Struktur. Diese Interrupt-Struktur ist jedoch in diesem Fall ein Bestandteil der 'CIA-Resource'-Struktur. Der 'is_Data'-Zeiger zeigt wieder auf die Resource-Struktur und '(*is_Code)()' zeigt auf eine Routine, die die Resource-Struktur verwaltet.

Von dieser Routine werden mit Hilfe der Struktur alle CIA Interrupts verwaltet, als da sind:

- Timer-A-Interrupt
- Timer-B-Interrupt
- Echtzeituhr-Alarm-Interrupt
- Serieller-Port- oder Tastatur-Interrupt
- Flagleitung-Interrupt

Die CIA-Resource-Struktur ist im Grunde eine Library-Struktur, die jedoch noch um eigene Einträge erweitert wurde.

Die CIA-Resource Struktur sieht wie folgt aus (Aufschlüsselungen der Unterstrukturen sind eingerückt dargestellt):

Offset	Bedeutung
0	struct Node lib_Node
0	Zeiger auf nächste Resource
4	Zeiger auf vorhergehende Resource
8	Node-type
9	Node-pri
10	Zeiger auf Resourcenname
14	UBYTE lib_Flags * \$00 *\
15	UBYTE lib_pab * \$00 *\
16	UWORD lib_NegSize * \$0018 *\
18	UWORD lib_PosSize * \$007C *\
20	UWORD lib_Version * \$0000 *\
22	UWORD lib_Revision * \$0000 *\
24	APTR lib_IdString * \$00000000 *\
28	ULONG lib_Sum * \$00000000 *\

```

32  UWORD  lib_OpenCnt           \* $0000 *\
34  APTR   CiaStartPtr
38  WORD   IntRequestBit
40  BYTE   IntEnableCia
41  BYTE   IntRegeestCia
42  struct Interrupt cia_Interrupt
      42  Zeiger auf nächsten Interrupt
      46  Zeiger auf vorherigen Interrupt
      50  Node-type
      51  Node-pri
      52  Zeiger auf Interruptname
      56  Zeiger auf Datenpuffer (hier Resource)
      60  Zeiger auf auszuführenden Code
64  struct IntVector Timer A
      CIAA und CIAB
      64  nicht initialisiert (00000000)
      68  nicht initialisiert (00000000)
      72  nicht initialisiert (00000000)
76  struct IntVector Timer B
      CIAA:
      76  Zeiger auf Datenpuffer
      80  Einsprung bei $FE9726
      84  Zeiger auf Interrupt-Struktur

      CIAB:
      76  nicht initialisiert (00000000)
      80  nicht initialisiert (00000000)
      84  nicht initialisiert (00000000)
88  struct IntVector TOD Alarm
      CIAA:
      88  nicht initialisiert (00000000)
      92  nicht initialisiert (00000000)
      96  nicht initialisiert (00000000)

      CIAB:
      88  Zeiger auf graphics.library
      92  Einsprung bei $FC6D68
      96  Zeiger auf Interrupt-Struktur
100 struct IntVector Seriell Data
      CIAA:
      100 Zeiger auf Keyboard.device
      104 Einsprung Tastenabfrage ($FE571C)
      108 Zeiger auf Interrupt-Struktur

```

```

      CIAB:
      100  nicht initialisiert (00000000)
      104  nicht initialisiert (00000000)
      108  nicht initialisiert (00000000)
112  struct IntVector Flag Leitung
      CIAA:
      112  nicht initialisiert (00000000)
      114  nicht initialisiert (00000000)
      118  nicht initialisiert (00000000)

      CIAB:
      112  Zeiger auf disk.resource
      114  Einsprung bei $FC4AB0
      118  Zeiger auf Interrupt-Struktur
```

Ein paar Erläuterungen dürften zu einigen Strukturunterpunkten noch zu machen sein.

CiaStartPtr

ist ein Zeiger auf den Beginn der CIA-Register. Bei CIAA ist dies \$BFE001 und bei CIAB \$BFD000.

IntRequestBit

ist das Bit, das im Interrupt-Request-Reg. gesetzt wird, wenn der Interrupt ausgelöst werden soll. Für CIAA ist dies \$0008 und für CIAB \$2000.

IntEnableCia

ist ein Byte, in dem vermerkt ist, welcher CIA-Interrupt erlaubt und nicht erlaubt ist.

IntRequestCia

ist ein Byte, in dem vermerkt wird, welcher CIA-Interrupt anliegt.

In den 'IntVektor'-Strukturen sind die Vektoren für die einzelnen CIA-Interrupts vermerkt. Nicht initialisierte Strukturen

werden vom Betriebssystem nicht benutzt und können für eigene Zwecke verwendet werden.

2.7.3.2 Die Verwaltung der Resource-Struktur

Nachdem die Struktur als solches bekannt ist, wird jetzt näher auf die Routine eingegangen, von der sie verwaltet wird. In A1 ist der Beginn der Resource-Struktur gespeichert.

Einsprung von CIAB:

fc4610	movem.l	A2/D2, -(A7)	D2 und A2 retten
fc4614	move.b	\$bfdd00, D2	Int.-Cont-Reg. nach D2 (CIAB)
fc461a	bra.s	\$fc4626	unbedingter Sprung

Einsprung von CIAA:

fc461c	movem.l	A2/D2, -(A7)	D2 und A2 retten
fc4620	move.b	\$bfed01, D2	Int.-Cont-Reg. nach D2 (CIAA)
fc4626	bclr	#7, D2	oberstes Bit löschen
fc462a	or.b	41(A1), D2	alte Bits mit neuen verknüpfen
fc462e	move.b	D2, 41(A1)	und Int.-Req.-Byte eintragen
fc4632	and.b	40(A1), D2	Req. mit Enable Byte verknüpfen
fc4636	beq.s	\$fc4658	verzweige, wenn Int. nicht zugelassen
fc4638	move.l	A1, A2	Zeiger auf Resource nach A2
fc463a	eor.b	D2, 41(A2)	Bit des abzuarbeitenden Interrupt im Reques-Reg. löschen
fc463e	lsr.b	#1, D2	Bit für Timer A Int. ins Carry
fc4640	bcs.s	\$fc465e	verzweige, wenn Timer A Int.
fc4642	lsr.b	#1, D2	Bit für Timer B Int. ins Carry
fc4644	bcs.s	\$fc4668	verzweige, wenn Timer B Int.
fc4646	beq.s	\$fc4658	verzweige, wenn keine Interrupts mehr vorhanden
fc4648	lsr.b	#1, D2	Bit für TOD-Alarm ins Carry
fc464a	bcs.s	\$fc4672	verzweige, wenn TOD-Alarm
fc464c	beq.s	\$fc4658	verzweige, wenn keine Interrupts mehr vorhanden
fc464e	lsr.b	#1, D2	Bit für Seriell-Data ins Carry
fc4650	bcs.s	\$fc467c	springe, wenn Seriell-Data Int.
fc4652	beq.s	\$fc4658	verzweige, wenn keine Interrupts mehr vorhanden
fc4654	lsr.b	#1, D2	Bit für Flag Int. ins Carry

fc4656 bcs.s	\$fc4686	verzweige, wenn Flag Interrupt
fc4658 movem.l	(A7)+,A2/D2	D2 und A2 wieder herstellen
fc465c rts		Rücksprung

Die Routine trägt jeden hereinkommenden Interrupt sofort in dem Interrupt-Request-Byte der 'Resource'-Struktur ein, sieht nach, ob der Interrupt erlaubt ist, und löscht, falls er erlaubt war, die entsprechenden Request-Bits wieder. Danach werden die Bits der anliegenden Interrupts systematisch ins Carry-Register geschoben und geprüft, ob sie erlaubt waren. Falls ja, wird der entsprechende Interrupt ausgeführt. Die Routinen zum Ausführen eines Interrupts werden jetzt anschließend beschrieben.

In A2 befindet sich der Zeiger auf die 'Resource'-Struktur.

Timer A Interrupt:

fc465e movem.l	64(A2),A5/A1	Einsprung und Datenzeiger holen
fc4664 jsr	(A5)	Einspringen
fc4666 bra.s	\$fc4642	zurück zur Int. Auswertung

Timer B Interrupt:

fc4668 movem.l	76(A2),A5/A1	Einsprung und Datenzeiger holen
fc466e jsr	(A5)	Einspringen
fc4670 bra.s	\$fc4648	zurück zur Int. Auswertung

TOD-Alarm Interrupt:

fc4672 movem.l	88(A2),A5/A1	Einsprung und Datenzeiger holen
fc4678 jsr	(A5)	Einspringen
fc467a bra.s	\$fc464e	zurück zur Int. Auswertung

Seriell-Data Interrupt:

fc467c movem.l	100(A2),A5/A1	Einsprung und Datenzeiger holen
fc4682 jsr	(A5)	Einspringen
fc4684 bra.s	\$fc4654	zurück zur Int. Auswertung

Flag Interrupt:

```
fc4686 movem.l 112(A2),A5/A1 Einsprung und Datenzeiger holen
fc468c jsr      (A5)           Einspringen
fc468e bra.s    $fc4658        zurück zur Int. auswertung
```

Wie eine Library, so verfügt auch die 'Resource'-Struktur über Funktionen, die ihre Verwaltung vereinfachen. Diese Funktionen sind von der Basisadresse aus mit negativen Offsets zu erreichen. Die CIA-Recource-Struktur verfügt über vier Funktionen. Diese sind:

Offset	Funktion
-6	SetInterrupt
-12	ClrInterrupt
-18	Clr/Set-Enabelbits
-24	ExecuteInterrupt

SetInterrupt

'IntVector'-Struktur nach Angaben setzen und Interrupt ermöglichen (Enabel-Bit). In A1 muß sich der Zeiger auf die 'Interrupt'-Struktur und in D0 Nummer des Interrupts befinden. Die Nummer 0 steht für Timer A Interrupt und 4 für Flag Interrupt. Mit dieser Funktion ist es nicht möglich, bereits initialisierte 'IntVektor'-Strukturen zu verändern. Zu diesem Zweck muß die Struktur zuvor mit der Funktion ClrInterrupt gelöscht werden. Der angegebene Interrupt wird gleichzeitig ermöglicht.

ClrInterrupt

'IntVektor'-Struktur löschen und Interrupt verhindern. In D0 befindet sich die Nummer der 'IntVector'-Struktur, die gelöscht werden soll. Der angegebene Interrupt wird gleichzeitig gesperrt.

Clr/Set-Enabelbits

Setzen der Interrupt-Enable-Bits im Hardware-Register sowie in der 'Resource'-Struktur. In D0 müssen die Bits gesetzt sein, die man zu ändern gedenkt. Bit 7 gibt an, ob diese Bits gelöscht oder gesetzt werden. Ist Bit 7 gelöscht, werden auch die angegebenen Bits im Interrupt-Enable-Register gelöscht. Mit D0 = \$03 würden beide Timerinterrupts gesperrt. Als Rückgabe erhält man in D0 den alten Stand des Hardware-Interrupt-Request-Register.

ExecuteInterrupt

Mit dieser Funktion ist es möglich, einen CIA-Interrupt mit der angegebenen Quelle softwaremäßig auszulösen. In D0 muß das Bit gesetzt sein, das im CIA-Interrupt-Request-Register für eine bestimmte Quelle steht. Zusätzlich muß noch Bit 7 von D0 gesetzt sein. Man müßte die Funktion mit D0 = \$81 aufrufen, um einen Timer A Interrupt zu erzeugen. Als Rückgabe erhält man in D0 den alten Stand des Hardware-Interrupt-Request-Register.

Der Aufruf der Funktionen kann nur fehlerfrei erfolgen, wenn der Zeiger auf die 'Resource'-Struktur in A6 steht. Mit

```
move.b #$02,d0
move.l ResourceBase,a6
jsr -18(a6)
```

würde der Timer B Interrupt gesperrt.

Als nächstes folgen die Assembler-Listings der einzelnen Funktionen.

SetInterrupt:

fc4690	moveq	#\$00,D1	D1 löschen
fc4692	move.b	D0,D1	Int.-Nummer nach D1
fc4694	mulu	#\$000c,D1	richtigen Offset berechnen
fc4698	move.l	\$0004,A0	ExecBase nach A0
fc469c	move.w	#\$4000,\$dff09a	Disable-
fc46a4	addq.b	#1,294(A0)	Macro


```

fc46a8 lea      64(A6,D1.W),A0  Beginn der Struktur bestimmen
fc46ac move.l   8(A0),D1        war Struktur schon initialisiert?
fc46b0 bne.s    $fc46e2         verzweige, wenn ja
fc46b2 move.l   A1,8(A0)        *iv-Node setzen
fc46b6 move.l   18(A1),4(A0)     (*iv_Code)() setzen
fc46bc move.l   14(A1),0(A0)     iv_Data setzen
fc46c2 move.w   #$0080,D1       Clr/Set-Bit auf Set setzen
fc46c6 bset     D0,D1           zu setzendes Bit eintragen
fc46c8 move.w   D1,D0           Wert nach D0 Übertragen
fc46ca bsr.s    $fc470a         zur Funktion Clr/Set-Enablebit
fc46cc moveq    #$00,D0         D0 löschen
fc46ce move.l   $0004,A0        ExecBase nach A0
fc46d2 subq.b   #1,294(A0)
fc46d6 bge.s    $fc46e0         Enable-
fc46d8 move.w   #$c000,$dff09a  Macro
fc46e0 rts                                Rücksprung

fc46e2 move.l   D1,D0           nicht verwendeter
fc46e4 bra.s    $fc46ce         Programmteil

```

ClrInterrupt:

```

fc46e6 moveq    #$00,D1        D1 löschen
fc46e8 move.b   D0,D1          Strukturnummer nach D1
fc46ea mulu     #$000c,D1       Offset berechnen
fc46ee move.l   $0004,A0        ExecBase nach A0
fc46f2 move.w   #$4000,$dff09a  Disable-
fc46fa addq.b   #1,294(A0)      Macro
fc46fe lea      64(A6,D1.W),A0  Strukturposition bestimmen
fc4702 clr.l    8(A0)           Zeiger auf 'Interrupt'-Struktur
                                löschen, als Zeichen, daß Struktur leer
fc4706 moveq    #$00,D1        D1 löschen
fc4708 bra.s    $fc46c6         Enablebits löschen

```

Clr/Set-Enablebits:

```

fc470a move.l   $0004,A0        ExecBase nach A0
fc470e move.w   #$4000,$dff09a  Disable-
fc4716 addq.b   #1,294(A0)      Macro
fc471a move.l   34(A6),A0       CiaStartPtr nach A0
fc471e move.b   D0,3328(A0)     CIA-Int.-Con.-Reg. setzen
fc4722 lea      40(A6),A1       Zeiger auf IntEnableCia
fc4726 bra.s    $fc474c         Bits setzen

```


2.7.4 Beschreibung der Interrupt-Funktionen

SetIntVector

```
Interrupt = SetIntVector (intNum, int)
           D0                D0    A1
```

Offset: -162

Beschreibung: Diese Funktion initialisiert die 'IntVector'-Struktur zum Benutzen mit einem Interrupt-Handler. Als Rückgabeparameter erhält man einen Zeiger auf die zuvor für diesen Interrupt genutzte 'Interrupt'-Struktur.

Parameter:

intNum gibt die Nummer des Interrupts an, den man zu nutzen gedenkt. Mit der Nummer 1 beispielsweise würde der Interrupt-Handler für 'Diskblock abgearbeitet' initialisiert.

int ist ein Zeiger auf die initialisierte 'Interrupt'-Struktur.

Ergebnis:

Interrupt In D0 wird ein Zeiger auf die zuvor benutzte 'Interrupt'-Struktur übergeben.

AddIntServer

```
AddIntServer (intNum, int)
           D0    A1
```

Offset: -168

Beschreibung: Die Funktion hängt die angegebene 'Interrupt'-Struktur in die Interrupt-Server-Liste ein. Die Priorität, die in der 'Node'-Struktur angegeben ist, entscheidet, an welcher Stelle der Interrupt in die Liste eingefügt wird. Ein Interrupt mit

hoher Priorität wird vor einem Interrupt mit niedriger Priorität ausgeführt.

Parameter:

intNum gibt die Nummer des Interrupts an, den man zu nutzen gedenkt.

int ist ein Zeiger auf die initialisierte 'Interrupt'-Struktur.

RemIntServer

```
RemIntServer (intNum, int)
                D0    A1
```

Offset: -174

Beschreibung: Die angegebene Interrupt-Struktur wird aus der Interrupt-Server-Liste herausgenommen.

Parameter:

intNum gibt die Nummer des Interrupts an, den man zu nutzen gedenkt.

int ist ein Zeiger auf die initialisierte 'Interrupt'-Struktur.

Cause

```
Cause (interrupt)
                A1
```

Offset: -180

Beschreibung: Von einem Task oder einem Interrupt wird softwaremäßig ein anderer Interrupt ausgelöst. Die Priorität dieses Interrupts ist niedriger als die eines Hardware-Interrupts, jedoch höher als die eines Tasks und kann dazu verwandt werden, asynchrone Steuerprozesse zu übernehmen.

Parameter:

interrupt ist ein Zeiger auf die initialisierte Interrupt-Struktur.

2.7.5 Beispiel eines Interrupt-Servers

Nachdem die Interrupt-Programmierung theoretisch besprochen wurde, folgt jetzt noch ein Beispiel, um das neu erworbene Wissen in der praktischen Anwendung zu erproben.

Mit dem folgenden Programm, das nur für die Besitzer einer RAM-Erweiterung in dieser Form interessant ist, ist es möglich, über F10 das gesamte Fast-Memory zu belegen und über F9 wieder einzuschalten. F1 dient zum Ausschalten der Tastenabfrage. Das Programm kann so modifiziert werden, daß man auch andere Prozesse über bestimmte Tasten in Gang setzen kann.

```

memtype = $10004
anzData = 300
allocmem = -198
freemem = -210
addIntServer = -168
RemIntServer = -174
taste = $bfec01
pri = 100
Type = 2
intNum = 3
is_Data = 14
is_Code = 18
ln_Type = 8
ln
ln_Pri = 9
ln_Name = 10

move.l $4,a6
move.l #anzData,d0
move.l #memtype,d1      ;fast, CLR
jsr allocmem(a6)
tst.l d0
beq fehler
move.l d0,a2
add.l #32,d0             ;Datespeicheranfang
move.l d0,is_Data(a2)

```

```

        move.b #pri,ln_pri(a2)
        move.b #type,ln_Type(a2)
        move.l #Ende-Anfang+8,d0 ;Codegröße bestimmen
        jsr allocmem(a6)         ;Speicher für Programm
        tst.l d0 ;Fehler?
        bne ok1 ;nein
        move.l a2,a1             ;*Interrupt
        move.l #anzData,d0
        jsr freemem(a6)         ;Speicher freimachen
        jmp fehler
ok1:    move.l d0,a3
        move.l a3,is_Code(a2)
        move.l a2,a1             ;*Interrupt-Struct

        lea.l Anfang,a2
        move.l #Ende-Anfang,d0
l1:     move.b (a2)+,(a3)+
        dbf d0,l1

        move.l #intNum,d0        ;für Tasten Int
        jsr addIntServer(a6)
        rts

```

```
fehler: rts
```

; in A1 ist der Zeiger auf den Datenspeicher
; gebraucht werden dürfen nur d0,d1,a1,a5,a6

```

Anfang: move.l d0,a5             ;*nächsten Interrupt
        move.b taste,d0
        not d0
        ror.b #1,d0
        cmp.b #$59,d0 ;F10
        beq Speicheraus
        cmp.b #$58,d0 ;F9
        beq Speicherein
        cmp.b #$50,d0 ;F1
        beq Abfrageaus
fehler1: clr.l d0
        rts

```

Speicheraus:

```

        mem = $20004             ;memory-Type
        availmem = -216

```

```
        move.l a1,a5
        tst.l (a5)
        bne fehler1
        move.l #$ffffff,(a5)+
        move.l $4,a6
l6:      move.l #mem,d1
        jsr availmem(a6)
        tst.l d0
        beq end1
        move.l d0,(a5)+
        jsr allocmem(a6)
        tst.l d0
        beq end1
        move.l d0,(a5)+
        bra l6
end1:    clr.l (a5)
        bra blink
```

Speicherein:

```
        move.l a1,a5
        move.l $4,a6
        tst.l (a5)
        beq fehler1
        clr.l (a5)+
l7:      tst.l (a5)
        beq end2
        move.l (a5)+,d0
        move.l (a5)+,a1
        jsr freemem(a6)
        bra l7
end2:    bra blink
```

Abfrageaus:

```
        move.l 4(a5),a1           ;Interrupt nach a1
        move.l #intNum,d0
        move.l $4,a6
        jsr RemIntServer(a6)
        bra blink

blink:   move.l #$2000,d0
l5:      move.w d0,$dff180
        sub.l #$01,d0
```

```
bne l5  
rts
```

Ende:

Mit diesem Programm wird eine Interrupt-Struktur eröffnet und in den Tastatur-Interrupt eingehängt. Da die Priorität unserer Interrupt-Struktur höher ist als die, die die Tastatur abfragt, wird unser Interrupt zuerst ausgeführt. Hier werden nun bestimmte Tasten abgefragt. Sollte F10 gedrückt worden sein, wird das gesamte Fast-Memory belegt und die Zeiger auf die belegten Bereiche mit ihren Längen im Interrupt-Datenpuffer abgelegt. Wird nun wieder F9 gedrückt, werden die im Interrupt-Datenpuffer vermerkten Zeiger benutzt, um den Speicher wieder freizugeben. Beim Drücken von F1 wird die Interrupt-Struktur wieder aus der Interrupt-Server-Liste entfernt, wodurch keine Tastenabfrage mehr erfolgen kann.

Als Signal, daß eine Taste gedrückt wurde und der Befehl ausgeführt werden konnte, blinkt der Bildschirm kurz auf.

2.8 Die ExecBase-Struktur

Die ExecBase-Struktur ist die Hauptstruktur von Exec, in der alle wichtigen Parameter vermerkt sind, beispielsweise welcher Task gerade läuft. Die Basisadresse der Struktur ist gleichzeitig die Basisadresse der Execlibrary und läßt sich folglich aus C heraus mit 'SysBase' adressieren. 'SysBase' ist eine standardisierte Variable, mit der sich die Position der Execlibrary ermitteln läßt.

Um aus Assembler auf die Struktur zugreifen zu können, muß man erst ihre Basisadresse bestimmen, welche immer in Speicherstelle \$000004 abgelegt ist.

Mit `move.l $4,a6` wird A6 die Basisadresse der Execlibrary bzw. der ExecBase-Struktur übergeben.

Da diese Struktur beim Einschalten von der Reset-Routine initialisiert wird, ist ihre Basisadresse immer gleich. Eine Ver-

schiebung kommt nur zustande, wenn die Größe oder die Lage des RAM-Speichers verändert wird. Nach dieser Veränderung ist ihre Position aber wiederum immer konstant.

Bei einem Amiga mit 512 KByte RAM ist die Position der ExecBase-Struktur \$676, sofern Kickstartversion 1.2 verwendet wird. Beim Ausbau auf 1MB verschiebt sich die ExecBase-Struktur auf \$C00276, was jedoch nur stimmt, wenn das zusätzliche RAM ab \$C00000 liegt.

Die Struktur hat folgendes Aussehen:

Offset		Adresse für		
DEZ	HEX	512KB	1MB	
				struct ExecBase {
0	\$000	\$676	\$C00276	struct Library LibNode;
34	\$022	\$698	\$C00298	UWORD SoftVer;
36	\$024	\$69A	\$C0029A	WORD LowMemChkSum;
38	\$026	\$69C	\$C0029C	ULONG ChkBase;
42	\$02A	\$6A0	\$C002A0	APTR ColdCapture;
46	\$02E	\$6A4	\$C002A4	APTR CoolCapture;
50	\$032	\$6A8	\$C002A8	APTR WarmCapture;
54	\$036	\$6AC	\$C002AC	APTR SysStkUpper;
58	\$03A	\$6B0	\$C002B0	APTR SysStkLower;
62	\$03E	\$6B4	\$C002B4	ULONG MaxLocMem;
66	\$042	\$6B8	\$C002B8	APTR DebugEntry;
70	\$046	\$6BC	\$C002BC	APTR DebugData;
74	\$04A	\$6C0	\$C002C0	APTR AlertData;
78	\$04E	\$6C4	\$C002C4	APTR MaxExtMem;
82	\$052	\$6C8	\$C002C8	UWORD ChkSum;
				struct IntVector IntVects[16];
84	\$054	\$6CA	\$C002CA	Serielle Ausgabe Pri1
96	\$060	\$6D6	\$C002D6	Disk Block Pri1
108	\$06C	\$6E2	\$C002E2	Softinterrupt Pri1
120	\$078	\$6EE	\$C002EE	CIAA Pri2
132	\$084	\$6FA	\$C002FA	Copper Pri3
144	\$090	\$706	\$C00306	Rasterstrahl Pri3
156	\$09C	\$712	\$C00312	Blitter fertig Pri3
168	\$0A8	\$71E	\$C0031E	Audiokanal0 Pri4
180	\$0B4	\$72A	\$C0032A	Audiokanal1 Pri4
192	\$0C0	\$736	\$C00336	Audiokanal2 Pri4
204	\$0CC	\$742	\$C00342	Audiokanal3 Pri4
216	\$0D8	\$74E	\$C0034E	Serielle Eingabe Pri5

```

228 $0E4 $75A $C0035A    Disksynchronisation    Pri5
240 $0F0 $766 $C00366    CIAB                    Pri6
252 $0FC $772 $C00372    Interner Interrupt    Pri6
264 $108 $77E $C0037E    NMI                    Pri7
276 $114 $78A $C0038A    struct Task *ThisTask;
280 $118 $78E $C0038E    ULONG    IdleCount;
284 $11C $792 $C00392    ULONG    DispCount;
288 $120 $796 $C00396    UWORD    Quantum;
290 $122 $798 $C00398    UWORD    Elapsed;
292 $124 $79A $C0039A    UWORD    SysFlags;
294 $126 $79C $C0039C    BYTE    IDNestCnt;
295 $127 $79D $C0039D    BYTE    TDNestCnt;
296 $128 $79E $C0039E    UWORD    AttnFlags;
298 $12A $7A0 $C003A0    UWORD    AttnResched;
300 $12C $7A2 $C003A2    APTR     ResModules;
304 $130 $7A6 $C003A6    APTR     TaskTrapCode;
308 $134 $7AA $C003AA    APTR     TaskExceptCode;
312 $138 $7AE $C003AE    APTR     TaskExitCode;
316 $13C $7B2 $C003B2    ULONG    TaskSigAlloc;
320 $140 $7B6 $C003B6    UWORD    TaskTrapAlloc;
322 $142 $7B8 $C003B8    struct List MemList;
336 $150 $7C6 $C003C6    struct List ResourceList;
350 $15E $7D4 $C003D4    struct List DeviceList;
364 $16C $7E2 $C003E2    struct List IntrList;
378 $17A $7F0 $C003F0    struct List LibList;
392 $188 $7FE $C003FE    struct List PortList;
406 $196 $80C $C0040C    struct List TaskReady;
420 $1A4 $81A $C0041A    struct List TaskWait;
434 $1B2 $828 $C00428    struct SoftIntList
                        SoftInts[5];

514 $202 $878 $C00478    LONG     LastAlert[4];
530 $212 $888 $C00488    UBYTE    VBlankFrequency;
531 $213 $889 $C00489    UBYTE    PowerSupplyFrequency;
532 $214 $88A $C0048A    struct List SemaphoreList;
546 $222 $898 $C00498    APTR     KickMemPtr;
550 $226 $89C $C0049C    APTR     KickTagPtr;
554 $22A $8A0 $C004A0    APTR     KickCheckSum;
558 $22E $8A4 $C004A4    UBYTE    ExecBaseReserved[10];
568 $22F $8AE $C004AE    UBYTE    ExecBaseNew-
                        Reserved[20];

};

#define SYSBASESIZE ((long)sizeof(struct ExecBase))

#define AFB_68010 0L
#define AFB_68020 1L
#define AFB_68881 4L

```

```
#define AFF_68010 (1L<<0)
#define AFF_68020 (1L<<1)
#define AFF_68881 (1L<<4)
#define AFB_RESERVED8 8L
#define AFB_RESERVED9 9L
```

LibNode - Offset 0

ist die 'Library'-Struktur der Exec-Library. Mit einer positiven Größe von \$24C und einer negativen Größe von \$276 Bytes. Anhand der positiven Größe kann man sehen, daß die 'ExecBase'-Struktur in Wirklichkeit eine etwas großgeratene Library-Struktur ist.

SoftVer - Offset 34

LowMemChkSum - Offset 36

kann vom Programmierer verwendet werden, um die Prüfsumme, die über den Bereich von Offset 34 bis 78 berechnet wird, auszugleichen, falls eigene Vektoren eingefügt wurden. Wie die eigentliche Prüfsumme berechnet wird, kann man sich unter ChkSum (Offset 82) ansehen.

ChkBase - Offset 38

wird gebraucht, um die Position von ExecBase beim Reset zu überprüfen. Die Position von Execbase wird mit ChkBase addiert, worauf das Ergebnis \$FFFFFFFF sein muß. Ist dies nicht der Fall, muß es zu einem größeren Fehler gekommen sein, weshalb die ExecBase-Struktur besser neu erstellt werden sollte. Ansonsten wird Zeit gespart und versucht, die Struktur nicht vollständig zu initialisieren.

ColdCapture - Offset 42

ist ein Vektor, der vom Programmierer benutzt werden kann, um während eines Resets in eine eigene Routine zu verzweigen. Ist der Vektor nicht benutzt, so steht er auf Null. Von der Reset-Routine wird erkannt, ob der Vektor gesetzt wurde und in die angegebene Routine verzweigt. In A5 wird die Rück-

sprungadresse abgelegt. Vor dem Einsprung über den Vektor wird dieser automatisch wieder auf Null gesetzt. Bis zu diesem Zeitpunkt ist bis auf das Sperren der Interrupts und des DMA-Zugriffs noch nichts Nennenswertes passiert. Man sollte bei der eigenen Routine keine Operationen durchführen, die mit dem Stapel arbeiten, da dieser noch nicht korrekt initialisiert ist. Das ist auch der Grund, warum der Rücksprung in A5 übergeben und die Routine nicht über einen 'JSR' aufgerufen wird.

CoolCapture - Offset 46

kann auch genutzt werden, um aus dem Reset in eine eigene Routine zu verzweigen. Der Unterschied zwischen ColdCapture und CoolCapture besteht darin, daß CoolCapture zu einem wesentlich späteren Zeitpunkt die eigene Routine aufruft. CoolCapture wird nicht von der Reset-Routine zurückgesetzt. Da der Stapel, der Speicher, die Exception-Tabelle und die Exec-Library hier schon initialisiert sind, ist der Vektor für die meisten Anwendungen besser geeignet als der ColdCapture-Vektor. Aus der eigenen Routine kann mit 'RTS' wieder in die Reset-Routine eingesprungen werden.

WarmCapture - Offset 50

ist ebenfalls ein Reset-Vektor, über den jedoch unseres Wissens nicht gesprungen wird.

SysStkUpper - Offset 54

gibt die oberste Grenze des Supervisor-Stacks an.

SysStkLower - Offset 58

gibt die untere Grenze des Supervisor-Stacks an.

MaxLocMem - Offset 62

gibt den maximal erreichbaren Chip-Memory-Bereich an, also 512KB oder \$80000 Bytes.

DebugEntry - Offset 66

ist ein Zeiger auf den Einsprung in den Debugger des Amiga.

DebugData - Offset 70

ist ein Zeiger auf den Datenpuffer des Debuggers. (Null)

*AlertData - Offset 74**MaxExtMem - Offset 78*

gibt die oberste Grenze des zur Verfügung stehenden Speichers an. Mit einer Speicher-Erweiterung auf 1MB ist dies \$C80000.

ChkSum - Offset 82

ist eine Prüfsumme über den Bereich von Offset 34 bis 78 und wird vor dem Einsprung in ColdCapture geprüft. Wenn eigene Vektoren in diesem Bereich eingebunden werden, muß die Prüfsumme neu berechnet, oder in LowChkSum ausgeglichen werden. Die Prüfsumme berechnet sich wie folgt:

fc0440	lea	34(A6),A0	Zeiger auf Anfang setzen
fc0444	move.w	#\$0016,D0	Anzahl der Worte -1 in Zähler
fc0448	add.w	(A0)+,D1	Worte addieren
fc044a	dbf	D0,\$fc0448	Zähler verringern
fc044e	not.w	D1	negieren und
fc0450	move.w	D1,82(A6)	in ChkSum speichern

IntVects[0] - Offset 84

Interrupt bei serieller Ausgabe. Nach Reset nicht initialisiert.

IntVects[1] - Offset 96

Interrupt bei Beendigung der Übertragung des Diskblocks. Nach Reset ist dies ein Interrupt-Handler.

IntVects[2] - Offset 108

Soft-Interrupt. Für genauere Beschreibung siehe Kapitel 2.6

IntVects[3] - Offset 120

CIAA-Interrupt. Nach Reset dient der Interrupt hauptsächlich zur Tastaturabfrage. (Interrupt-Server)

IntVects[4] - Offset 132

Copper-Interrupt.

IntVects[5] - Offset 144

Interrupt, der ausgelöst wird, wenn der Rasterstrahl Rasterzeile Null passiert. Der Interrupt ist auch das Taktsignal für das Task-Switching. (Interrupt-Server)

IntVects[6] - Offset 156

Der Interrupt wird ausgelöst, wenn der Blitter mit seiner Arbeit fertig ist. (Interrupt-Handler)

IntVects[7] - Offset 168

Audiokanal 0 (Interrupt-Handler)

IntVects[8] - Offset 180

Audiokanal 1 (Interrupt-Handler)

IntVects[9] - Offset 192

Audiokanal 2 (Interrupt-Handler)

IntVects[10] - Offset 204

Audiokanal 3 (Interrupt-Handler)

IntVects[11] - Offset 216

Der Interrupt wird ausgelöst, wenn eine serielle Eingabe erfolgt ist. (nach Reset nicht initialisiert)

IntVects[12] - Offset 228

Der Interrupt wird ausgelöst, bei der Synchronisation der Diskette. (Interrupt-Handler)

IntVects[13] - Offset 240

Der Interrupt wird ausgelöst, wenn ein Interrupt von CIAB ausgelöst wird. (Interrupt-Server)

IntVects[14] - Offset 252

Der Interrupt kann nur softwaremäßig ausgelöst werden (nach Reset nicht initialisiert).

IntVects[15] - Offset 264

Nicht maskierbarer Interrupt. Der Interrupt wird zwar nicht verwendet, ist jedoch als Interrupt-Server initialisiert.

**ThisTask - Offset 276*

ist ein Zeiger auf die Task-Struktur, die gerade bearbeitet wird. Den Zeiger auf diese Task-Struktur kann man nicht mit einem Programm, das in einem Task läuft, auslesen und sinnvolle Werte erhalten, da man immer den Zeiger auf die eigene Task-Struktur erhält. Die einzige Möglichkeit, die man hat, sinnvolle Werte zu erhalten, ist den Wert aus einem Interrupt zu lesen. Hierfür bietet sich der Interrupt 5 (Rasterstrahl) an.

*idleCount - Offset 280**DispCount - Offset 284**Quantum - Offset 288*

*Elapsed - Offset 290**SysFlags - Offset 292*

In diesem Flag werden für das System wichtige Bits gesetzt.

Bit 5: 0 = Soft-Interrupt nicht erlaubt, 1 = erlaubt

IDNestCnt - Offset 294

gibt an, ob Interrupts erlaubt sind. Steht IDNestCnt auf \$FF (-1), sind Interrupts erlaubt, ansonsten wurden sie mit der Disable-Funktion gesperrt. Jedesmal, wenn die Disable-Funktion aufgerufen wird, wird IDNestCnt um Eins erhöht. Bei der Enable-Funktion wird IDNestCnt wieder verringert. Erst wenn IDNestCnt auf -1 steht, werden die Interrupts wieder ermöglicht (das Masterbit wird wieder gesetzt).

TDNestCnt - Offset 295

gibt an, ob die Forbid-Funktion aufgerufen wurde. Wurde sie aufgerufen, so wird TDNestCnt um Eins erhöht. Das Umschalten zu einem anderen Task ist erlaubt, wenn TDNestCnt nicht auf -1 steht. Durch die Permit-Funktion wird TDNestCnt wieder herabgezählt und auch das Umschalten auf einen anderen Task wieder ermöglicht, sofern TDNestCnt wieder auf -1 steht.

AttnFlags - Offset 296

gibt an, welche Prozessoren angeschlossen sind:

```
#define AFF_68010 (1L<<0)
#define AFF_68020 (1L<<1)
#define AFF_68881 (1L<<4)
```

AttnResched - Offset 298

Resmodules - Offset 300

ist ein Zeiger auf resistente Module. Dies sind Strukturen, über die Routinen von einer Routine ab \$fC0AF0 aufgerufen werden. Die Module werden bei einem Reset aufgerufen. Es steht auch eine Funktion zur Verfügung, mit der sich diese Module nach ihren Namen suchen lassen. Sie nennt sich 'FindResident'.

*TaskTrapCode - Offset 304**TaskExceptCode - Offset 308**TaskExitCode - Offset 312**TaskSigAlloc - Offset 316**TaskTrapAlloc - Offset 320**MemList - Offset 322*

ist ein Zeiger auf die Speicherliste, wo verzeichnet ist, welcher Speicherbereich frei und belegt ist.

ResourceList - Offset 336

ist eine List-Struktur, in der die Resource-Strukturen verkettet sind.

DeviceList - Offset 350

ist eine List-Struktur, in der die Resource-Strukturen verkettet sind.

IntList - Offset 364

Wird nicht verwendet

LibList - Offset 378

ist eine List-Struktur, in der die Library-Strukturen verkettet sind.

PortList - Offset 392

ist eine List-Struktur, in der die Port-Strukturen verkettet sind.

TaskReady - Offset 406

ist eine List-Struktur, in der die Task-Strukturen verkettet sind, die zur Zeit auf 'Ready' stehen.

TaskWait - Offset 420

ist eine List-Struktur, in der die Task-Strukturen verkettet sind, die zur Zeit auf 'Wait' stehen.

SoftInts[0] - Offset 434

ist eine List-Struktur, in der die Soft-Interrupts verkettet sind, die zur Zeit auf ihre Abarbeitung warten und die Priorität -32 haben.

SoftInts[1] - Offset 450

ist eine List-Struktur, in der die Soft-Interrupts verkettet sind, die zur Zeit auf ihre Abarbeitung warten und die Priorität -16 haben.

SoftInts[2] - Offset 466

ist eine List-Struktur, in der die Soft-Interrupts verkettet sind, die zur Zeit auf ihre Abarbeitung warten und die Priorität 0 haben.

SoftInts[3] - Offset 482

ist eine List-Struktur, in der die Soft-Interrupts verkettet sind, die zur Zeit auf ihre Abarbeitung warten und die Priorität +16 haben.

SoftInts[4] - Offset 498

ist eine List-Struktur, in der die Soft-Interrupts verkettet sind, die zur Zeit auf ihre Abarbeitung warten und die Priorität +32 haben.

LastAlert[4] - Offset 514

Hier werden die Daten für den Alert zwischengespeichert, nachdem sie von der Reset-Routine geholt wurden.

VBlankFrequency - Offset 530

gibt an, mit welcher Frequenz der Rasterstrahl ein Bild aufbaut. (50Hz)

PowerSupplyFrequency - Offset 531

gibt an welche Frequenz die Netzspannung hat, mit der der Amiga versorgt wird. (50Hz)

SemaphoreList - Offset 532

ist eine List-Struktur, in der alle Semaphore-Strukturen, sofern sie benutzt werden, verkettet sind.

KickMemPtr - Offset 546

ist ein Zeiger auf eine MemList-Struktur, deren Speicher bei einem Reset wieder belegt wird.

KickTagPtr - Offset 550

ist ein Zeiger auf eine Resident-Tabelle, die bei der Erstellung der Haupt-Resident-Tabelle eingebunden wird.

KickChecksum - Offset 554

ist die Prüfsumme die durch die Funktion SumKickData() berechnet wird.

ExecBaseReserved[10] - Offset 558

sind 10 Bytes, die für die ExecBase-Struktur reserviert sind, um Exec die Möglichkeit zu geben, beliebige Werte dort zu speichern. (nicht genutzt)

ExecBaseNewReserved[20] - Offset 568

sind 20 Bytes, die für die ExecBase-Struktur reserviert sind, um Exec die Möglichkeit zu geben, beliebige Werte dort zu speichern. (nicht genutzt)

2.9 Reset-Routine und resetfeste Programme

In diesem Kapitel wird der Frage nachgegangen, wie die Reset-Routine genau abläuft, wie die Speichergröße bestimmt wird und ob es möglich ist, resetfeste Programme zu schreiben.

2.9.1 Dokumentation der Reset-Routine

```
fc00d2 lea    $040000,A7    Stackpointer setzen
fc00d8 move.l #$00020000,D0 Wert für Warteschleife
fc00de subq.l #1,D0        Wert verringern
fc00e0 bgt.s  $fc00de       verzweige, wenn nicht abgezählt
fc00e2 lea    -228(PC)(=$fc0000),A0 Zeiger auf Kickstart-
                                ROM setzen
fc00e6 lea    $f00000,A1    Vergleichswert laden
fc00ec cmpa.l A1,A0        liegt Reset ab $F00000
fc00ee beq.s  $fc00fe       verzweige, wenn ja
fc00f0 lea    12(PC)(=$fc00fe),A5 Zeiger auf Programm
                                Fortsetzung stellen
fc00f4 cmpi.w #$1111,(A1)   liegt ab $F00000 Modul?
fc00f8 bne.s  $fc00fe       verzweige, wenn nein
fc00fa jmp    2(A1)         sonst einspringen
fc00fe move.b #$03,$bfe201  Port auf Ausgang schalten
fc0106 move.b #$02,$bfe001  LED Ausschalten
fc010e lea    $dff000,A4    Zeiger auf Chipadressen
fc0114 move.w #$7fff,D0     Wert laden
fc0118 move.w D0,154(A4)    Alle Interrupts sperren
fc011c move.w D0,156(A4)    Interruptanfragen löschen
fc0120 move.w D0,150(A4)    DMA Zugriff sperren
```

```

fc0124 move.w  #0200,256(A4)
fc012a move.w  #0000,272(A4)
fc0130 move.w  #0444,384(A4)    Farbe setzen
fc0136 move.w  #0008,A0        Zeiger auf Exeptions setzen
fc013a move.w  #002d,D1        Zähler für Anzahl der Vektoren
fc013e lea     1140(PC)(=$fc05b4),A1  Zeiger auf Harderror
                                      Routine setzen
fc0142 move.l  A1,(A0)+        Exceptions eintragen
fc0144 dbf     D1,$fc0142      verzweige, wenn nicht fertig
fc0148 bra.l   $fc30c4         Guru prüfen

```

Prüfen, ob Reset durch Guru entstanden, falls ja, Guru-Nummer in D7 und Memory-Ptr. in D6 übernehmen. Ansonsten wird in D6 \$FFFFFFF geladen. Daraufhin wird der Reset weitergeführt.

```

fc014c move.l  $0004,D0        ExecBase holen
fc0150 btst    #0,D0           ExecBase auf gerader Adresse?
fc0154 bne.s   $fc01ce         Fehler, wenn ungerade
fc0156 move.l  D0,A6           Execbase nach D0
fc0158 add.l   38(A6),D0       ChkBase hinzuaddieren
fc015c not.l   D0              Ergebnis invertieren
fc015e bne.s   $fc01ce         verzweige, wenn Fehler

```

Der Fehler tritt auf, wenn der Zeiger auf die ExecBase-Struktur falsch ist.

```

fc0160 moveq   #00,D1          D1 für Prüfsumme löschen
fc0162 lea     34(A6),A0        Zeiger auf ChkBase stellen
fc0166 moveq   #18,D0          Wert für Schleifen Zähler
fc0168 add.w   (A0)+,D1         Prüfsumme bilden
fc016a dbf     D0,$fc0168      verzweige, bis Summe gebildet
fc016e not.w   D1              Ergebnis invertieren
fc0170 bne.s   $fc01ce         Fehler, wenn nicht Null
fc0172 move.l  42(A6),D0       ColdCapture nach D0
fc0176 beq.s   $fc0184         verzweige, wenn nicht gesetzt
fc0178 move.l  D0,A0           Pointer nach A0
fc017a lea     8(PC)(=$fc0184),A5  Zeiger auf Fortsetzung
fc017e clr.l   42(A6)          ColdCapture löschen
fc0182 jmp     (A0)            einspringen
fc0184 bchg    #1,$bfe001      LED anschalten
fc018c move.l  -382(PC)(=$fc0010),D0  Kick. Version mit
fc0190 cmp.l   20(A6),D0       der der Execlib. vergleichen
fc0194 bne.s   $fc01ce         Fehler, wenn Versionen ungleich
fc0196 move.l  62(A6),A3       obere Grenze des Chipmemory

```

```

fc019a cmpa.l  #$00080000,A3 512KB Chipmemory?
fc01a0 bhi.s   $fc01ce        Fehler, wenn größer
fc01a2 cmpa.l  #$00040000,A3 256KB Chipmemory?
fc01a8 bcs.s   $fc01ce        Fehler, wenn kleiner
fc01aa move.l  78(A6),A4      MaxExtMem nach A4
fc01ae move.l  A4,D0          prüfe ob externer Speicher da
fc01b0 beq.l   $fc0240        verzweige, wenn nicht vorhanden
fc01b4 cmpa.l  #$00dc0000,A4  obere Grenze bei $DC0000?
fc01ba bhi.s   $fc01ce        Fehler, wenn höher
fc01bc cmpa.l  #$00c40000,A4  obere Grenze bei $C40000?
fc01c2 bcs.s   $fc01ce        Fehler, wenn niedriger
fc01c4 move.l  A4,D0          unsinnig, da A4 = D0
fc01c6 andi.l  #$0003ffff,D0  liegt Grenze auf glatter Adr.?
fc01cc beq.s   $fc0240        ja, sonst Fehler

```

Hier beginnt der Teil der Routine, der nur angesprungen wird, wenn etwas mit der Initialisierung der vorgefundenen ExecBase-Struktur nicht stimmt. Sie muß folglich neu initialisiert werden.

```

fc01ce lea     $0400,A6        unterster möglicher RAM-Bereich
fc01d2 suba.w  #$fd8a,A6      Adresse von ExecBase bestimmen,
                               für den Fall daß kein Fast-Mem.

fc01d6 lea     $c00000,A0      unterster Fast-Mem-Bereich
fc01dc lea     $dc0000,A1      oberste mögliche RAM-Grenze
fc01e2 lea     6(PC)(=$fc01ea),A5 Zeiger auf Fortsetzung
fc01e6 bra.l   $fc061a        obere Speichergrenze holen

```

In der hier aufgerufenen Routine wird festgestellt, wo die obere Grenze des Fast-RAMs liegt. Sie übergibt den Zeiger auf das RAM-Ende in A4. Steht kein Fast-RAM zur Verfügung, wird eine Null in A4 zurückgegeben. Diese Selbsterkennung des Fastmemory funktioniert jedoch nur, wenn das RAM auch ab \$C00000 liegt. Für die Besitzer eines Amiga 1000 bietet sich die Möglichkeit, durch Änderung der Kickstart-Diskette eine RAM-Erweiterung, die nicht ab \$C00000 liegt, autokonfigurierend arbeiten zu lassen.

```

fc01ea move.l  A4,D0          Fast-RAM vorhanden?
fc01ec beq.s   $fc0208        verzweige, wenn nicht vorhanden
fc01ee move.l  #$00c00000,A6  untere Grenze nach A6
fc01f4 suba.w  #$fd8a,A6      Position von ExecBase ermitteln
fc01f8 move.l  A4,D0          obere Grenze nach D0

```

```
fc01fa lea    $c00000,A0    untere Grenze nach A0
fc0200 lea    6(PC)(=$fc0208),A5  Zeiger auf Fortsetzung
fc0204 bra.l  $fc0602        Speicher löschen
```

In der Routine wird der Fast-Memory-Bereich mit Nullen aufgefüllt.

```
fc0208 lea    $0000,A0        unterste RAM-Grenze
fc020c lea    $200000,A1      oberste Grenze des zusammen-
                             hängenden Speichers
fc0212 lea    6(PC)(=$fc021a),A5  Zeiger auf Fortsetzung
fc0216 bra.l  $fc0592        untere Speichergröße holen
```

Der untere Speicherbereich darf im Bereich von \$0000 bis \$200000 liegen. In der Routine wird geprüft, wie groß der untere zusammenhängende Speicher ist. Die Speichergröße wird in A3 zurückgegeben.

```
fc021a cmpa.l  #$00040000,A3 Bereich kleiner 256KB?
fc0220 bcs.s   $fc0238        ja, dann Fehler, Hard-Reset
fc0222 move.l  #$00000000,$0000 $0000 löschen
fc022a move.l  A3,D0          obere Speichergrenze nach D0
fc022c lea     $00c0,A0        untere Grenze festlegen
fc0230 lea     14(PC)(=$fc0240),A5  Zeiger auf Fortsetzung
fc0234 bra.l   $fc0602        unteren Speicher löschen
```

Die Routine wurde bereits für das Löschen des Fast-RAMs benutzt. Sie löscht den Bereich von \$00C0 bis zur oberen Speichergrenze des unteren zusammenhängenden Speichers.

```
fc0238 move.w  #$00c0,D0      Bildfarbe bei Reset
fc023c bra.l   $fc05b8        Hard-Reset (LED 11*Blinken)
```

Bei diesem Reset-Einsprung blinkt die LED 11x auf, worauf in das Boot-ROM eingesprungen und der Reset erneut gestartet wird. Zu dieser Routine wird auch verzweigt, wenn eine Exception-Bedingung während des ersten Teils des Resets auftritt.

```
fc0240 lea     $dff000,A0      Zeiger auf Chipadressen
fc0246 move.w  #$7fff,150(A0)  DMA-Zugriffe sperren
fc024c move.w  #$0200,256(A0)
fc0252 move.w  #$0000,272(A0)
```

```

fc0258 move.w  #0888,384(A0)    Bildfarbe setzen
fc025e lea      84(A6),A0        Zeiger auf ersten IntVector
fc0262 movem.l  546(A6),D4-D2    KickMemPrt, KickTagPrt,
                                KickChecksum speichern
fc0268 moveq    #00,D0          D0 löschen
fc026a move.w   #007d,D1        Zähler setzen
fc026e move.l   D0,(A0)+        von A0 ab ExecBase löschen
fc0270 dbf      D1,$fc026e      verzweige, wenn nicht fertig
fc0274 movem.l  D4-D2,546(A6)    KickMemPrt, KickTagPrt und
                                KickChecksum setzen
fc027a move.l   A6,$0004        ExecBase Zeiger setzen
fc027e move.l   A6,D0           Zeiger nach D0
fc0280 not.l    D0              ChkBase berechnen
fc0282 move.l   D0,38(A6)       ChkBase eintragen
fc0286 move.l   A4,D0           oberste RAM-Grenze nach D0
fc0288 bne.s    $fc028c        verzweige, wenn Fast-RAM zur
                                Verfügung steht
fc028a move.l   A3,D0           sonst obere Chip-Ram Grenze
fc028c move.l   D0,A7           als System-Stack setzen
fc028e move.l   D0,54(A6)       und eintragen
fc0292 sub.l    #00001800,D0    Länge des Stacks abziehen
fc0298 move.l   D0,58(A6)       und als untere Grenze setzen
fc029c move.l   A3,62(A6)       Grenze des Chip-RAMs setzen
fc02a0 move.l   A4,78(A6)       Grenze des Fast-RAMs setzen
fc02a4 bsr.l    $fc30e4         Last Alert eintragen

```

Die Werte, die bei \$FC0148 geholt und in D6 und D7 gespeichert wurden, wieder an den dafür vorgesehenen Platz (LastAlert (Offset 514)) schreiben.

```

fc02a8 bsr.l    $fc0546        Prozessor-Test

```

Die Routine testet, welche Prozessoren angeschlossen sind. Erkennt wird: 68000, 68010, 68020, 68881. Entsprechend der erkannten Prozessoren werden die Bits in D0 gesetzt.

```

fc02ac or.w     D0,296(A6)      Bits in AttnFlags setzen
fc02b0 lea      32(PC)=(A0),A1  Zeiger auf Tabelle

fc02b4 move.w   (A1)+,D0        Offset nach D0
fc02b6 beq.l    $fc033e        Ende, wenn kein Offset mehr
fc02ba lea      0(A6,D0.W),A0   Zeiger auf Position setzen
fc02be move.l   A0,(A0)        Listenkopf eintragen
fc02c0 addq.l   #4,(A0)        auf lh_Tail zeigen lassen

```


fc02c2	clr.l	4(A0)	lh_Tail löschen
fc02c6	move.l	A0,8(A0)	lh_TainPred setzen
fc02ca	move.w	(A1)+,D0	lh_Type holen
fc02cc	move.b	D0,12(A0)	und setzen
fc02d0	bra.s	\$fc02b4	unbedingter Sprung

Die soeben beschriebene Teilroutine trägt folgende List-Strukturen wieder in ExecBase ein:

MemList
 ResourceList
 DeviceList
 LibList
 PortList
 TaskReady
 TaskWait
 InterList
 SoftIntList (alle 5)
 SemaphoreList

fc02d2

.
 .
 .

Tabellen für die Erstellung der Listen

.
 .

fc030c

.
 .

Werte für die Erstellung der Execlibrary-Struktur

.
 .

fc0326

.
 .

ASCII-Texte: Chip Memory

.
 .

fc0332

.
 .

ASCII-Text: Fast Memory

.
.
fc033e

```
fc033e lea      11380(PC)(=$fc2fb4),A0  Zeiger auf Task Tarp-
                                         Code setzen
fc0342 move.l   A0,304(A6)      in TaskTrapCode eintragen
fc0346 move.l   A0,308(A6)      in TaskExceptCode eintragen
fc034a move.l   #$00fc1cec,312(A6) TaskExitCode eintragen
fc0352 move.l   #$0000ffff,316(A6) TaskSigAlloc eintragen
fc035a move.w   #$8000,320(A6)   TaskTarpAlloc eintragen
fc0360 lea      8(A6),A1        Zeiger auf LibNode.ln_Type
fc0364 lea      -90(PC)(=$fc030c),A0  Zeiger auf Tabelle
fc0368 moveq    #$0c,D0        Zähler setzen
fc036a move.w   (A0)+,(A1)+     Execlibrary-Struktur erstellen
fc036c dbf      D0,$fc036a      verzweige, wenn nicht fertig
fc0370 move.l   A6,A0          Zeiger auf ExecBase nach A0
fc0372 lea      5836(PC)(=$fc1a40),A1  Zeiger auf Tabelle
fc0376 move.l   A1,A2
fc0378 bsr.l    $fc1576         Funktion: MakeFunktion()
```

In der hier angesprungenen Routine wird die Execlibrary erstellt. Die Tabelle dazu steht ab \$FC1A40. In D0 wird die Länge der Library zurückgegeben.

```
fc037c move.w   D0,16(A6)      Länge der Library eintragen
fc0380 move.l   A4,D0          ist Fast-RAM vorhanden?
fc0382 beq.s    $fc03a8        verzweige, wenn kein Fast-RAM
fc0384 lea      588(A6),A0      Zeiger auf ExecBase Ende
fc0388 lea      -88(PC)(=$fc0332),A1 String Fast Mem.
fc038c moveq    #$00,D2        Priorität des MemHeader
fc038e move.w   #$0005,D1      Memory-Attribute (Public, Fast)
fc0392 move.l   A4,D0          Zeiger auf Fast-RAM Ende
fc0394 sub.l    A0,D0          ExecBase-Struktur abziehen
fc0396 sub.l    #$00001800,D0  SysStack abziehen
fc039c bsr.l    $fc19ea        MemHeader-Struktur erstellen
```

Die Routine ab \$FC19EA erstellt eine MemHeader-Struktur mit den angegebenen Daten. In D0 steht die Größe des zur Verfügung stehenden Speicherbereichs.

```
fc03a0 lea      $0400,A0       Speicheranfang Chip-RAM
fc03a4 moveq    #$00,D0        D0 löschen
fc03a6 bra.s    $fc03b2        unbedingter Sprung
```

```

fc03a8 lea      588(A6),A0      Zeiger auf ExecBase Ende
fc03ac move.l   #$ffffe800,D0
fc03b2 move.w   #$0003,D1      Memory-Attribute (Public, Chip)
fc03b6 move.l   A0,A2          Zeiger auf RAM-Anfang
fc03b8 lea      -148(PC)=(=$fc0326),A1 String Chip Mem.
fc03bc moveq    #$f6,D2        Priorität des MemHeaders
fc03be add.l    A3,D0           Errechnung des
fc03c0 sub.l    A0,D0           effektiven Speicherbereichs
fc03c2 bsr.l    $fc19ea        MemHeader-Struktur erstellen
fc03c6 move.l   A6,A1          ExecBase nach A1
fc03c8 bsr.l    $fc140c        Library-Prüfsumme berechnen

```

Die Execlibrary wird in die LibList verkettet und ihre Prüfsumme berechnet.

```

fc03cc lea      938(PC)=(=$fc0778),A0 Zeiger auf Exceptions
fc03d0 move.l   A0,A1          Zeiger auf exceptions nach A1
fc03d2 move.w   #$0008,A2      Zeiger auf Ziel nach A2
fc03d6 bra.s    $fc03de        unbedingter Sprung
fc03d8 lea      0(A0,D0.W),A3   Adresse berechnen
fc03dc move.l   A3,(A2)+       und eintragen
fc03de move.w   (A1)+,D0        Offset holen
fc03e0 bne.s    $fc03d8        verzweige, wenn nicht fertig
fc03e2 move.w   296(A6),D0      AttnFlags holen
fc03e6 btst     #0,D0          68010 eingesetzt?
fc03ea beq.s    $fc041e        verzweige, wenn nicht vorhanden
fc03ec lea      1166(PC)=(=$fc087c),A0 Zeiger auf neue Traps
fc03f0 move.w   #$0008,A1      Zeiger auf Ziel
fc03f4 move.l   A0,(A1)+       Exceptions eintragen
fc03f6 move.l   A0,(A1)+       Exceptions eintragen
fc03f8 move.l   #$00fc08ba,-28(A6) Erweiterung eintragen
fc0400 move.l   #$42c04e75,-528(A6) Erweiterung eintragen
fc0408 btst     #4,D0          68881 eingebaut?
fc040c beq.s    $fc041e        verzweige, wenn nicht vorhanden
fc040e move.l   #$00fc108a,-52(A6) Erweiterung eintragen
fc0416 move.l   #$00fc10e8,-58(A6) Erweiterung eintragen
fc041e bsr.l    $fc125c        Interrupt-Struktur eintragen
fc0422 lea      $dff000,A0      Zeiger auf Chipadressen
fc0428 move.w   #$8200,150(A0) Blitter-DMA erlauben
fc042e move.w   #$c000,154(A0) Interrupts erlauben
fc0434 move.w   #$ffff,294(A6) IDNestCnt löschen
fc043a bsr.l    $fc22fa        Debugger installieren
fc043e moveq    #$00,D1        D1 löschen
fc0440 lea      34(A6),A0      Zeiger auf SoftVer
fc0444 move.w   #$0016,D0      Zähler nach D0

```

```

fc0448 add.w  (A0)+,D1      Prüfsumme berechnen
fc044a dbf     D0,$fc0448    verzweige, wenn nicht fertig
fc044e not.w   D1           Wert invertieren
fc0450 move.w  D1,82(A6)     und in ChkSum speichern
fc0454 lea     118(PC)(=$fc04cc),A0 Zeiger auf MemList-Str.
fc0458 bsr.l   $fc191e      AllcoEntry()

```

In der Routine wird eine MemList-Struktur erstellt und \$1024 Bytes für den Task und dessen Stack reserviert.

```

fc045c move.l  D0,A2        Zeiger auf MemList-Struktur
fc045e lea     4112(A2),A0   Zeiger auf Start für Task
fc0462 lea     8(A0),A1      MemEntry hinzurechnen
fc0466 addi.l  #$00000010,D0 MemList hinzuaddieren
fc046c move.l  D0,58(A1)     SpLower setzen
fc0470 move.l  A0,62(A1)     SpUpper setzen
fc0474 move.l  A0,54(A1)     SpReg setzen
fc0478 move    A0,USP        auch als Stapel setzen
fc047a clr.b   9(A1)         Pri löschen
fc047e move.b  $0001,8(A1)   in tc_Type Wert für Task
fc0484 move.l  #$00fc00a8,10(A1) Zeiger auf Name

```

Der Name des Tasks ist 'exec.library'. Er wird später wieder entfernt.

```

fc048c lea     74(A1),A0     Zeiger auf tc_MemEntry
fc0490 move.l  A0,(A0)       Liste
fc0492 addq.l  #4,(A0)       für MemEntries
fc0494 clr.l   4(A0)         erstellen
fc0498 move.l  A0,8(A0)
fc049c exg     A2,A1         Memlist- und Task-Zeiger
                             tauschen
fc049e bsr.l   $fc15d8       AddHead()
fc04a2 exg     A2,A1         zurücktauschen
fc04a4 move.l  A1,276(A6)    Task als ThisTask eintragen
fc04a8 suba.l  A2,A2         initPc
fc04aa move.l  A2,A3         und finalPC löschen
fc04ac bsr.l   $fc1c48       AddTask()
fc04b0 move.l  276(A6),A1    Zeiger auf Task nach A1
fc04b4 move.b  #$02,15(A1)   tc_State auf RUN setzen
fc04ba bsr.l   $fc1600       Remove() Task aus Liste

```

Der Task wird auf Running gesetzt und aus der TaskReady-Liste entfernt, was bedeutet, daß das ab jetzt laufende Programm als Task abgearbeitet wird.

```
fc04be andi.w  #0000,SR    Alle Interrupts sperren
fc04c2 addq.b  #1,295(A6)  SysFlag setzen
fc04c6 jsr     -138(A6)    Permit() (Task zulassen)
fc04ca bra.s   $fc0500     unbedingter Sprung
```

```
fc04cc
```

```
.
```

```
.
```

Daten für MemoryList-Struktur

```
.
```

```
.
```

```
fc04fe
```

```
fc0500 lea     -30(PC)(=$fc04e4),A0
fc0504 bsr.l   $fc0900     Resident-Struktur suchen
```

In der Routine werden alle in ROM befindlichen Resident-Strukturen gesucht und der Zeiger auf diese Strukturen in einer Tabelle hintereinander abgelegt. Zusätzlich wird auch geprüft, ob die Pointer KickMemPtr, KickTagPtr und KickChecksum in der ExecBase-Struktur gesetzt wurden. Sollte dies der Fall sein, so werden die angegebenen Speicherbereiche belegt, und die angegebenen Resident-Struktur in die erwähnte Tabelle übernommen, die mit Null abgeschlossen wird. Die Reihenfolge der Einträge der Tabelle richtet sich nach der Priorität der Resident-Strukturen. Der Zeiger auf die Tabelle wird in ResModules gespeichert.

```
fc0508 move.l  D0,300(A6)
fc050c bclr    #1,$bfe001   LED-Anschalten (ist schon an)
fc0514 move.l  46(A6),D0    CoolCapture holen
fc0518 beq.s   $fc051e      verzweige, wenn nicht gesetzt
fc051a move.l  D0,A0        CoolCapture nach A0
fc051c jsr     (A0)         einspringen
fc051e moveq   #01,D0       startClass setzen
fc0520 moveq   #00,D1       Version setzen
fc0522 bsr.l   $fc0af0      InitCode(), Resident-Strukturen bearbeiten
```

In den folgenden Programmteil wird nicht mehr eingesprungen.

```
fc0526 move.l 50(A6),D0      WarmCapture holen
fc052a beq.s  $fc0530        verzweige, wenn nicht gesetzt
fc052c move.l D0,A0          WarmCapture nach A0
fc052e jsr   (A0)            einspringen
fc0530 moveq  #$0d,D0        Wert für Zähler setzen
fc0532 clr.l  -(A7)          Stack löschen
fc0534 dbf   D0,$fc0532      verzweige, wenn nicht fertig
fc0538 movem.l (A7)+,A5-A0/D7-D0
fc053c jsr   -114(A6)        Einsprung in Debugger
fc0540 move.l $0004,A6       ExecBase nach A6
fc0544 bra.s  $fc053c        unbedingter Sprung
```

2.9.2 Resident-Strukturen

Um besser zu verstehen, wie es möglich ist, resetfeste Module "einzubauen", muß zuvor besprochen werden, was Resident-Strukturen sind und wie sie verarbeitet werden.

Resident-Strukturen sind Strukturen, die sich im Betriebssystem befinden und nach denen bei einem Reset gesucht wird. Gesucht werden diese Strukturen nach ihrem Erkennungscode, der am Anfang der Struktur vermerkt ist. Die Positionen aller gefundenen Resident-Strukturen werden in einer Tabelle zusammengefaßt, und ein Zeiger auf diese Tabelle wird daraufhin in ResModuls (in der ExecBase-Struktur) vermerkt.

Beim weiteren Verlauf des Resets wird der Zeiger auf die zuvor erstellte Tabelle wieder geholt und in die InitCode()-Funktion verzweigt. Mit Hilfe der Zeiger in der Tabelle werden die Resident-Strukturen wieder gefunden.

In dieser Funktion wird der Sinn der Resident-Strukturen erst klar. In einer solchen Struktur ist unter anderem ein Zeiger gespeichert, der abhängig von den Flags, die in der Resident-Struktur vermerkt sind, auf eine Tabelle für die Register beim Funktionsaufruf MakeLib() oder auf ein auszuführendes Programm zeigt. Einfach gesagt, hat man mit einer Resident-Struktur die Möglichkeit, in ein eigenes Programm zu verzweigen oder die Funktion MakeLib() aufzurufen.

Wenn man die Funktion MakeLib() aufrufen lassen will, hat man noch weitere Variationen zur Verfügung. Man kann entscheiden, ob die mit der MakeLib() Funktion erstellte Struktur mit AddLibrary() in die Library-Liste, mit AddDevice() in die Device-Liste oder mit AddResource() in die Resource-Liste eingefügt werden soll. Diese Möglichkeiten bestehen, weil mit der Funktion MakeLib() aufgrund der Ähnlichkeit der Strukturen sowohl Library-, Device-, als auch Resource-Strukturen erstellt werden können.

Eine Resident-Struktur hat folgendes Aussehen:

```
struct Resident {  
0   UWORD rt_MatchWord;  
2   struct Resident *rt_MatchTag;  
6   APTR  rt_EndSkip;  
10  UBYTE rt_Flags;  
11  UBYTE rt_Version;  
12  UBYTE rt_Type;  
13  BYTE  rt_Pri;  
14  char  *rt_Name;  
18  char  *rt_IdString  
22  APTR  rt_Init;  
};
```

```
#define RTC_MATCHWORD 0x4AFCL  
#define RTF_AUTOINIT (1L<<7)  
#define RTF_COLDSTART (1L<<0)  
#define RTM_WHEN 3L  
#define RTW_NEVER 0L  
#define RTW_COLDSTART 1L  
#endif
```

rt_MatchWord

ist ein Wort, nach dem die Struktur im Speicher erkannt wird. Nach diesem Erkennungswort wird beim Reset gesucht, wenn die Resident-Strukturen gesucht werden. Das Wort muß den Wert \$4AFC haben, damit die Struktur gefunden wird

rt_MatchTag

ist ein Zeiger auf die Struktur selbst und wird ebenfalls zur Erkennung der Struktur im Speicher verwandt. Nachdem das MatchWord gefunden wurde, wird geprüft, ob das darauffolgende Lanwort ein Zeiger auf die eigene Struktur ist. Ist das der Fall, wird eine Resident-Struktur erkannt.

rt_EndSkip

ist ein Zeiger auf das Ende der Struktur. Mit diesem Zeiger ist es möglich, die Struktur beliebig lang zu machen und wichtige Daten in ihr zu vermerken.

rt_Flags

gibt je nach Setzen der Bits an, ob die Resident-Struktur überhaupt bearbeitet werden soll, und wenn ja, ob nur der angegebene Befehl oder der Einsprung in das angegebene Programm erfolgt. Ist das oberste Bit (Bit 7) gelöscht, so bedeutet dies, daß in das in der Struktur vermerkte Programm eingesprungen wird. Ist es gesetzt, so zeigt *rt_Init* auf eine Tabelle für die Register, die für das Ausführen der Funktion *MakeLib()* nötig sind.

rt_Version

gibt an, um welche Version der Struktur es sich handelt.

rt_Type

gibt an, welcher Befehl ausgeführt werden soll.

rt_Name

ist ein Zeiger auf den Namen der Struktur.

rt_idString

ist ein Zeiger auf den String, der nähere Erläuterungen zu der Struktur gibt.

rt_Init

ist ein Zeiger auf das auszuführende Programm oder ein Zeiger auf die Tabelle für die zu ladenden Registerinhalte beim Aufruf der MakeLib()-Funktion. Soll die MakeLib()-Funktion aufgerufen werden, so müssen in der Tabelle folgende Register in angegebener Reihenfolge eingetragen sein:

D0 = DataSize
 D1 = CodeSize
 A0 = FuncInit
 A1 = StructInit
 A2 = LibInit

Wie eine Tabelle der Zeiger auf Resident-Strukturen aussieht, auf die ResModuls (in der ExecBase-Struktur) zeigt, läßt sich mit der folgenden Tabelle verdeutlichen. Die Endmarkierung der Tabelle ist das letzte Langwort, das den Wert Null hat.

Die Tabelle wird normalerweise von der Reset-Routine erstellt.

Sollte das oberste Bit (Bit 31) eines Langworts in der Tabelle gesetzt sein, so bedeutet das, daß das Langwort ohne Berücksichtigung des obersten Bits ein Zeiger auf die Fortsetzung der Tabelle ist.

```
00fc 00b6 00fc 4afc 00fe 4880 00fe 4fe4
00fc 450c 00fc 4794 00fe 4774 00fe 49cc
00fc 5378 00fe 502e 00fe 507a 00fe 90ec
00fc 34cc 00fe 50c6 00fe 0d90 00fe 510e
00fe 98e4 00fd 3f5c 00fc 323a 00fe 424c
00fe b400 00ff 425a 00fe 8884 0000 0000
```

Sehen wir uns von der Tabelle ausgehend einmal die zweite Resident-Struktur an. Sie sieht wie folgt aus:

```
fc4afc 4afc 00fc 4afc 00fc 516c 8121 096e 00fc .....
fc4b0c 4b48 00fc 4b16 00fc 4b38 6578 7061 6e73 .....expans
fc4b1c 696f 6e20 3333 2e31 3231 2028 3420 4d61 ion 33.121 (4 Ma
fc4b2c 7920 3139 3836 290d 0a00 0000 0000 01c8 y 1986).....
fc4b3c 00fc 4b86 00fc 4b5a 00fc 4bee 6578 7061 .....expa
fc4b4c 6e73 696f 6e2e 6c69 6272 6172 7900 e000 nsion.library...
```

Um die Initialisierung der Struktur besser nachvollziehen zu können, sind die entsprechenden Werte in der folgenden Struktur noch einmal aufgeführt.

```

struct Resident {
0   UWORD rt_MatchWord;           $4AFC
2   struct Resident *rt_MatchTag; $FC4AFC
6   APTR rt_EndSkip;              $FC516C
10  UBYTE rt_Flags;               %10000001
11  UBYTE rt_Version;             33
12  UBYTE rt_Type;                Library
13  BYTE rt_Pri;                  110
14  char *rt_Name;                expansion.library
18  char *rt_IdString             $FC4B16
22  APTR rt_Init;                 $FC4B38
};

```

Das `rt_Flag` Byte ist auf `%10000001` gesetzt. Das oberste Bit ist also gesetzt, weshalb das `rt_Init` auf Daten für die Register zeigt, die für den Aufruf der Funktion `AddLibrary()` gebraucht werden. Hier wird die `AddLibrary()`-Funktion aufgerufen, da der Typ der Resident-Struktur 'Library' (`NT_LIBRARY = 09`) ist.

Es stehen folgende Möglichkeiten zur Verfügung:

Type	Funktionsaufruf
03 = Device	AddDevice()
08 = Resource	AddResource()
09 = Library	AddLibrary()

Zum Untersuchen der Resident-Struktur ist die Funktion `'InitCode()'` zuständig. Beim Aufruf werden die zwei Parameter `'StartClass'` und `'Version'` übergeben. `'Version'` legt fest, daß nur Resident-Strukturen ausgeführt werden, deren Version gleich oder über der angegebenen ist. Der Wert, der in `'StartClass'` übergeben wird, wird mit `rt_Flags` UND-verknüpft. Sollte das Ergebnis ungleich Null sein, wird die Struktur ausgeführt und somit die `'initResident'`-Funktion aufgerufen. Mit `'StartClass'` und `rt_Flags` wird also festgelegt, welche Resident-Strukturen beim Aufruf der `'InitCode'`-Funktion ausgeführt werden.

Bei einem Reset wird die 'InitClass'-Funktion mit den Parametern 'StartClass' = 01 und 'Version' = 00 ausgeführt. Es werden somit nur die Resident-Strukturen ausgeführt, deren Bit 0 in `rt_Flags` gesetzt ist.

Damit Sie sich noch ansehen können, wie die eben beschriebenen Routinen genau ablaufen, folgen hier die Assemblerlistings der Funktionen.

InitCode

InitCode(StartClass, Version)

D0

D1

```

fc0af0 movem.l A2/D3-D2, -(A7)  Register retten
fc0af4 move.l  300(A6), A2      Zeiger auf ResModuls
fc0af8 move.b  D0, D2          StartClass nach D2
fc0afa move.b  D1, D3          versin nach D3
fc0afc move.l  (A2)+, D0        Zeiger aus Tabelle holen
fc0afe beq.s   $fc0b22          verzweige, wenn Endmarkierung
fc0b00 bgt.s   $fc0b0a          verzweige, wenn oberstes Bit
                                nicht gesetzt (Bit 31)
fc0b02 bclr    #31, D0          sonst Bit 31 löschen
fc0b06 move.l  D0, A2          Zeiger als neuen Zeiger auf
                                die Tabelle speichern
fc0b08 bra.s   $fc0afc          unbedingter Sprung
fc0b0a move.l  D0, A1          Zeiger auf Resident nach A1
fc0b0c cmp.b   11(A1), D3      Version vergleichen
fc0b10 bgt.s   $fc0afc          Resident-Version zu alt
fc0b12 move.b  10(A1), D0      rt_Flags holen
fc0b16 and.b   D2, D0          mit StartClass verknüpfen
fc0b18 beq.s   $fc0afc          verzweige, wenn nicht starten
fc0b1a moveq   #0, D1          segList auf Null
fc0b1c jsr     -102(A6)        InitResident()
fc0b20 bra.s   $fc0afc          unbedingter Sprung
fc0b22 movem.l (A7)+, A2/D3-D2 Register zurückholen
fc0b26 rts                                Rücksprung

```

InitResident

InitResident(resident, segList)

A1 D1

fc0b28	btst	#7,10(A1)	Bit 7 von rt_Flags testen
fc0b2e	bne.s	\$fc0b3c	nicht gesetzt, dann Befehl ausführen
fc0b30	move.l	22(A1),A1	sonst Einsprung holen
fc0b34	moveq	#00,D0	D0 löschen
fc0b36	move.l	D1,A0	segList nach A0
fc0b38	jsr	(A1)	Einspringen
fc0b3a	bra.s	\$fc0b7e	unbedingter Sprung
fc0b3c	movem.l	A2-A1,-(A7)	Register retten
fc0b40	move.l	22(A1),A1	Zeiger auf Register holen
fc0b44	movem.l	(A1),A2-A0/D0	Register für Funktion holen
fc0b48	jsr	-84(A6)	MakeLib()
fc0b4c	movem.l	(A7)+,A2/A0	Register zurückholen
fc0b50	move.l	D0,-(A7)	Rückmeldung holen
fc0b52	beq.s	\$fc0b7c	Fehler, dann Ende
fc0b54	move.l	D0,A1	Zeiger auf Library nach A1
fc0b56	move.b	12(A0),D0	rt_type nach D0
fc0b5a	cmpi.b	#03,D0	rt_type = Device?
fc0b5e	bne.s	\$fc0b66	verzweige, wenn nicht Device
fc0b60	jsr	-432(A6)	sonst AddDevice()
fc0b64	bra.s	\$fc0b7c	unbedingter Sprung
fc0b66	cmpi.b	#09,D0	rt_type = Library?
fc0b6a	bne.s	\$fc0b72	verzweige, wenn nicht Library
fc0b6c	jsr	-396(A6)	sonst AddLibrary()
fc0b70	bra.s	\$fc0b7c	unbedingter Sprung
fc0b72	cmpi.b	#08,D0	rt_type = Resource?
fc0b76	bne.s	\$fc0b7c	verzweige, wenn nicht Resource
fc0b78	jsr	-486(A6)	sonst AddResource
fc0b7c	move.l	(A7)+,D0	D0 zurückholen
fc0b7e	rts		Rücksprung

2.9.3 Resetfeste Programme und Strukturen

Nachdem jetzt alle Voraussetzungen geschaffen worden sind, soll jetzt besprochen werden, wie man resetfeste Programme und Strukturen programmieren kann.

Dazu gibt es zwei Möglichkeiten. Zum einen kann man Über den Vektor ColdCapture und CoolCapture in das eigene Programme verzweigen, zum anderen besteht die Möglichkeit, mit Hilfe der Vektoren KickMemPrt und KickTagPrt Speicherbereiche erneut zu belegen und Resident-Struktur in die Resident-Vektor-Tabelle einzufügen. All diese Vektoren sind in der ExecBase-Struktur zu finden.

Es kommt auf den Zweck an, welche Möglichkeit man nutzt. Um lediglich einen Reset zu sperren, ist der ColdCapture-Einsprung wohl am günstigsten. Sie müssen lediglich den Vektor auf eine Routine legen, die den ColdCaprute-Vektor erneut setzt und daraufhin in einer Endlosschleife läuft.

Beim Initialisieren des ColdCapture-Vektors müssen Sie zusätzlich noch die Prüfsumme, die über die ersten ExecBase Vektoren gebildet wird, neu berechnen. Eine Routine zum Sperren des Resets hätte beispielweise folgendes Aussehen:

```
run:
allocMem = -198
require = 1
ColdCapture = 42

        move.l $4,a6
        move.l #Ende-Anfang,d0
        move.l #require,d1
        jsr allocMem(a6)
        tst.l d0
        beq fehler
        move.l d0,a1
        move.l a1,ColdCapture(a6)
        move.w #Ende-Anfang,d0
        lea.l Anfang,a0
l1:      move.b (a0)+,(a1)+
        dbf d0,l1
```

; Prüfsumme neu berechnen

```

        clr.l d1
        lea.l 34(a6),a0
        move.w #$16,d0
l2:      add.w (a0)+,d1
        dbf d0,l2
        not.w d1
        move.w d1,82(a6)
fehler: rts

```

; Routine die bei Reset ausgeführt wird

```

Anfang: lea.l anfang(pc),a0
        move.l $4,a6
        move.l a0,ColdCapture(a6)
l3:      jmp l3(pc)
Ende:

```

Das Programm belegt zuerst den benötigten Speicher, kopiert das Programm, das bei einem Reset gestartet werden soll, in diesen Bereich, trägt die Anfangsadresse des Programms in ColdCapture ein und berechnet die Prüfsumme neu.

Das Neusetzen des ColdCapture-Vektors ist nötig, da dieser von der Reset-Routine gelöscht wird.

Wenn das Programm gestartet ist und ein Reset erfolgte, rettet Sie nur noch das Ausschalten des Rechners.

Mit ColdCapture wird sehr früh aus der Reset-Routine in das eigene Programm verzweigt. Sie können sich den Aussprung in der dokumentierten Reset-Routine ab \$FC0172 ansehen. Bis zu diesem Zeitpunkt des Aussprungs ist noch nicht viel geschehen. Es wurde lediglich die Bildschirmfarbe auf schwarz geschaltet, alle Interrupts und DMA-Zugriffe gesperrt und die Prüfsumme der ExecBase-Struktur überprüft.

Sollte ColdCapture gesetzt worden sein, so wird der Vektor von der Reset-Routine wieder gelöscht, die Adresse, wo die Reset-Routine weitergeführt werden soll, in A5 übergeben und in die Routine, die in ColdCapture angegeben ist, verzweigt. In der

eigenen Routine darf nicht mit dem Stapel gearbeitet werden, also auch kein Unterprogrammaufruf erfolgen, da der Stapel noch nicht initialisiert wurde.

Mit JMP (A5) wird die Reset-Routine wieder weitergeführt.

Der andere Vektor, der genutzt werden kann, um im Verlauf eines Resets aus der Reset-Routine in ein eigenes Programm zu verzweigen, heißt CoolCapture. Wenn aus der Reset-Routine in das eigene Programm verzweigt wird, ist der Speicher bereits wieder organisiert, die ExecBase-Struktur und die Execlibrary erstellt, die Interrupts eingerichtet und die Exceptions wieder auf ihre richtigen Werte zurückgesetzt.

Über CoolCapture wird mit einem 'JSR (A0)' eingesprungen und deshalb auch normalerweise mit 'RTS' abgeschlossen. Der Vektor kann verwandt werden, um eine Speicher-Erweiterung anzumelden, die nicht automatisch angemeldet wird. Der Aussprung aus der Reset-Routine ist bei \$FC051C.

Die Vorgehensweise ist gleich der beim ColdCapture-Einsprung. Der Vektor wird eingetragen und daraufhin die Prüfsumme der ExecBase-Struktur neu berechnet. Wie die Prüfsumme berechnet wird, können Sie sich in dem Beispielprogramm für den ColdCapture-Einsprung ansehen.

Die Verwendung der KickSumData()-Funktion:

Die beste Möglichkeit, resetfeste Programme und Strukturen zu erhalten, ist, sie mit Hilfe der ExecBase-Einträge KickMemPtr, KickTagPtr und KickChecksum zu erzeugen.

Durch Benutzung der Einträge kann man beliebige Speicherbereiche mit deren alten Werten erneut belegen und eigene Resident-Strukturen in die Tabelle der Zeiger auf Resident-Strukturen einfügen. Um das zu erreichen, muß KickMemPtr auf eine MemList-Struktur zeigen, deren Einträge während eines Resets wieder belegt werden. KickTapPtr ist ein Zeiger auf eine Resident-Tabelle, die aussieht wie die, auf die ResModuls in der ExecBase-Struktur zeigt. Die in der Tabelle vermerkten

Resident-Strukturen werden abhängig von ihrer Priorität in die zu erstellende Tabelle aller Resident-Strukturen, die später ausgeführt werden, eingebunden.

Diese beiden Zeiger werden jedoch nur verwendet, wenn KickChecksum (die Prüfsumme über die Resident-Tabelle und die MemList-Strukturen) richtig ist. Zur Berechnung der Prüfsumme dient eine Execlibrary-Funktion mit dem Namen KickSumData().

KickSumData

```
Summe = KickSumData()  
D0
```

Offset: -612

Beschreibung: Die Funktion berechnet die Prüfsumme über die in KickMemPtr angegebene MemList-Struktur sowie die in KickTagPtr angegebene Resident-Tabelle. Das Ergebnis der Berechnung wird in D0 zurückgegeben.

Wenn man bestimmte Speicherbereiche nach einem Reset ohne Verlust der Daten wieder belegt haben möchte, so muß man diese Bereiche in einer MemList-Struktur vermerken und mit AllocEntry() belegen. Die eigene MemList-Struktur muß auch mitbelegt sein. Es ist auch möglich, mehrere MemList-Strukturen miteinander zu verketteten.

Wenn man bei einem Reset zusätzlich noch eigene Programme ausführen will, so erstellt man Resident-Strukturen, mit denen man die gewünschten Programme aufruft. Die Zeiger auf die Resident-Strukturen müssen in einer Tabelle zusammengefasst werden, die mit Null abgeschlossen wird. Der Speicherbereich der Resident-Strukturen der auszuführenden Programme sowie die Resident-Tabelle müssen ebenfalls mit einer MemList-Struktur belegt worden sein, und in die Liste des bei einem Resets zu belegenden Speichers, wie zuvor beschrieben, eingefügt werden. Der Zeiger auf die Resident-Tabelle wird in KickTagPtr eingetragen.

Nachdem diese Schritte unternommen wurden, wird die KickSumData()-Funktion aufgerufen und die berechnete Prüfsumme in KickChecksum gespeichert.

Jetzt sind die angegebenen Speicherbereiche resetgeschützt, und die in den Resident-Strukturen vermerkten Programme werden bei einem Reset ausgeführt.

Zur Orientierung, welche Prioritäten die eigenen Resident-Strukturen haben müssen, um an der gewünschten Stelle ausgeführt zu werden, folgt nun eine Tabelle, die zeigt, welche Resident-Strukturen mit welchen Prioritäten bei einem Reset ausgeführt werden.

Pri.	Resident-Pos.	Beschreibung
120	\$FC00B6	Exec-Lib. erstellen (nicht erlaubt)
110	\$FC4AFC	Exception-Lib. erstellen
100	\$FE4880	Potgo-Lib erstellen
80	\$FC450C	CIA-Resources erstellen
70	\$FC4794	Disk-Resource erstellen
70	\$FE4774	Misc-Resource erstellen
70	\$FE49CC	RAM-Lib. erstellen (nicht erlaubt)
65	\$FC5378	Graphics-Lib. erstellen
60	\$FE502E	Keyboard-Device erstellen
60	\$FE507A	Gameport-Device erstellen
50	\$FE90EC	Timer-Device erstellen
40	\$FC34CC	Audio-Device erstellen
40	\$FE50C6	Input-Device erstellen
31	\$FE0D90	Layers-Lib. erstellen
20	\$FE510E	Console-Device erstellen
20	\$FE98E4	Trackdisk-Device erstellen
10	\$FD3F5C	Intuition-Lib. erstellen
5	\$FC323A	Gurus, wenn vorhanden, ausgeben
0	\$FE424C	Math-Lib erstellen
0	\$FEB400	Workbench-Task, (nicht erlaubt)
0	\$FF425A	DOS-Lib erstellen (nicht erlaubt)
-60	\$FE8884	Einspringen in Bootvorgang

Nach der letzten Resident-Struktur kann keine eigene mehr ausgeführt werden, da nach dem Aufruf dieser Struktur nicht mehr zu der InitCode()-Funktion zurückgekehrt wird. Eine niedrigere Priorität als -60 ist also unsinnig.

2.9.4 Ein richtiges NoFastMem

Zum Abschluß dieses Kapitels zeigen wir noch eine weitere Variante, das Fast-Memory anzuschalten.

Die herkömmlichen NoFastMem-Routinen belegen einfach das gesamte Fast-Memory, so daß weitere Programme ins Chip-Memory geladen werden müssen. Wenn man jedoch einmal nachdenkt, wird man feststellen, daß alle wichtigen Strukturen wie beispielweise die ExecBase-Struktur oder die Interrupts bereits während eines Resets initialisiert werden und deshalb immer im Fast-Memory liegen. Das kann der Grund dafür sein, daß manche Programme trotz Ausschalten des Fast-RAMs auf die herkömmliche Weise nicht laufen. Eine andere Alternative bietet sich jedoch mit der Verwendung der Reset-Routine.

Man braucht lediglich ein Programm, das alle wichtigen Schritte, die während eines Resets gemacht werden, ebenfalls unternimmt, und dann an einer geeigneten Stelle in die Reset-Routine wieder einzuspringt. In seiner eigenen Routine kann man ohne weiteres dem Reset "vorgaukeln", daß kein Fast-RAM vorhanden ist. Springt man nun an der Stelle in den Reset wieder ein, wo alle Strukturen wieder neu initialisiert werden und täuscht man dem Reset vor, daß kein Fast-RAM vorhanden ist, so werden alle Strukturen im Chip-RAM initialisiert.

Diese Täuschung bleibt solange erhalten, bis die ExecBase-Struktur von einem Programm zerstört wird, denn dann wird erneut nachgesehen, wieviel Speicher wirklich vorhanden ist. Ansonsten wird der Vermerk, daß kein Fast-RAM zu finden ist, bei jedem weiteren Reset aus der ExecBase-Struktur geholt und alle Strukturen danach initialisiert.

Ein solches Programm hat folgendes Aussehen:

```
move.b #$03,$bfe201      ;Port auf Ausgang
move.b #$02,$bfe001      ;LED ausschalten
lea.l $dff000,a4
move.w #$7fff,d0
move.w d0,154(a4)
move.w d0,156(a4)        DMA-Zugriffe sperren
```

```
move.w d0,150(a4)
move.w #$0200,256(a4)
move.w #$0000,272(a4)
move.w #$0444,384(a4)

lea.l $400,a6
suba.w #$fd8a,a6          ;ExecBase-Basis ermitteln
lea.l $c00000,a0
lea.l $dc0000,a1
move.l #$00,a4            ;kein Fast-RAM vorhanden
move.l a4,d0
move.l #$40000,a7         ;Hilfsstackpointer nach A7
move.l $ffffffff,d6       ;Wert für keinen Alert gefunden
jmp $fc0208               ;Einsprung in Reset
```


3. Das AmigaDOS

Den für den Anwender wichtigsten Teil des Amiga-Betriebssystems stellt das DOS dar, was für Disc-Operating-System steht. Seine Aufgabe ist es, sämtliche Ein- und Ausgabe-Funktionen zu erledigen, zum Beispiel Disketten-Operationen oder Tastatur-Eingaben. Die vielfältigen Funktionen, die dazu nötig sind, werden dem Benutzer in Form einer Bibliothek zugänglich gemacht, die ähnlich der EXEC-Bibliothek aufgebaut ist.

3.1 Die DOS-Bibliothek

Ebenso wie die EXEC-Bibliothek befindet sich die DOS-Bibliothek nicht als Datei auf der Diskette wie z.B. die Intuition-Library. Dennoch muß sie vor der Benutzung erst einmal geöffnet werden, damit man den Zugriff zum DOS bekommt. Beim diesem Vorgang bekommt das öffnende Programm über eine Zeigertabelle Zugang zu den DOS-Funktionen.

3.1.1 Laden der DOS.LIBRARY

Will ein Programm eine Funktion der DOS-Bibliothek nutzen, so muß es diese erst einmal öffnen. Dafür wird eine Funktion des EXEC verwendet, die den Namen `OldOpenLibrary` trägt. Dieser Funktion wird ein Zeiger auf den Namen der Bibliothek übergeben, der in Kleinbuchstaben vorliegen und mit einem Null-Byte abgeschlossen sein muß. Es kann auch die Funktion `OpenLibrary` verwendet werden, der ein weiterer Parameter übergeben werden muß: die gewünschte Version der Bibliothek. Ist die Versionsnummer der Bibliothek größer oder gleich dieser Nummer, so wird diese geöffnet. Aus diesem Grund wird üblicherweise hier eine Null eingetragen, damit die Version keine Rolle spielt.

In C gestaltet sich dies recht einfach. Durch die Zeile

```
DOSBase = OpenLibrary ("dos.library",0);
```

erhält man von EXEC in DOSBase einen Zeiger auf die DOS-Bibliothek übergeben, mit dem man dann die DOS-Funktionen aufrufen kann. Der Zeiger muß dabei nicht weiterverwendet werden, da der C-Compiler dies übernimmt. Lediglich die Kontrolle, ob das DOS ordnungsgemäß geöffnet wurde, kann durch Überprüfen des Rückgabewertes geschehen: er ist Null, wenn ein Fehler aufgetreten ist. Dies kann etwa so aussehen:

```
if (DOSBase == 0) exit (DOS_OPEN_ERROR);
```

In Maschinensprache ist dies nicht ganz so einfach, aber dennoch leicht zu überschauen. Das Öffnen der DOS-Bibliothek wird etwa folgendermaßen programmiert:

```
EXEC_Base      = 4
OldOpenLibrary = -408

        move.l  EXEC_Base,a6      ;Zeiger auf EXEC-Base in A6
        lea     DOS_Name,a1       ;Zeiger auf Bibliotheksnamen
        jsr     OldOpenLibrary(a6) ;Bibliothek öffnen
        move.l  d0,DOS_Base       ;Zeiger auf DOS-Base retten
        beq     error            ;Fehler aufgetreten
        ...

error:
        ...

DOS_Base:    dc.l  0              ;Platz für DOS-Basis
DOS_Name:    dc.b  "dos.library",0
```

Der in D0 erhaltene Zeiger wird für jeden folgenden Aufruf einer DOS-Funktion benötigt. Hat das Öffnen der Bibliothek nicht funktioniert, so wird in D0 eine Null zurückgegeben und in diesem Programm in die Fehlerbehandlungs-Routine 'error' verzweigt.

Die DOS-Bibliothek wird also vom obigen Programm durch den Zeiger verfügbar gemacht. Sie ist ähnlich wie die EXEC-Bibliothek aufgebaut und wird daher auf die gleiche Art und Weise bearbeitet. Die Einsprung-Adressen der einzelnen Funktionen liegen unterhalb der in `DOS_Base` liegenden Basisadresse und werden daher auch mit negativen Offsets aufgerufen.

3.1.2 Funktionsaufruf und Parameterübergabe

Für den Aufruf einer DOS-Funktion werden außer der Adresse der Funktion selbst meist noch einige Parameter benötigt, die übergeben werden müssen. Diese Parameter werden in den Prozessor-Datenregistern D1 bis D4 übergeben.

Ein Beispiel: Um ein einfaches Fenster zu öffnen, wird die DOS-Funktion `Open()` verwendet. Es werden folgende Parameter benötigt:

- Ein Zeiger auf den Namen der zu öffnenden Datei, der mit einem Null-Byte endet, im Register D1. Für unser Beispiel wird als Name die Fensterdefinition CON: mit den entsprechenden Parametern eingesetzt.
- Die Angabe des Zugriffs-Modus wird in D2 verlangt. Dieser Modus gibt an, ob es sich um eine neu anzulegende oder eine bereits existierende Datei handelt. Für das im Beispiel zu öffnende Fenster wird der Modus 'alt' übergeben, damit aus dem Fenster auch gelesen werden kann.

Das Maschinenprogramm für dieses Beispiel sieht dann etwa folgendermaßen aus:

```
Open      = -30
Mode_old  = 1005
```

```
...
move.l    #FileName,d1      ;Zeiger auf Datei-Definition
move      #Mode_old,d2      ;Modus: alt
move.l    DOS_Base,a6       ;DOS-Basisadresse in A6
```

```

jsr      Open(a6)           ;Datei (Fenster) öffnen
move.l   d0,ConHandle       ;Dateihandle-Zeiger retten
beq      error              ;Fehler aufgetreten!
...

```

```

ConHandle: dc.l 0           ;Platz für Dateihandle
FileName:  dc.b "CON:10/10/620/200/** Test-Fenster **",0

```

Auf die genaue Verwendung des Standard-Kanals CON: wird in einem späteren Kapitel eingegangen.

3.1.3 Die DOS-Funktionen

In diesem Kapitel werden nun alle DOS-Funktionen aufgeführt. Die Offsets werden ebenso angegeben wie die Register, in denen die verschiedenen Parameter übergeben werden müssen.

1) allgemeine Ein-/Ausgabe-Funktionen

Open

```

Handle = Open (Name, Modus)
D0      -30   D1   D2

```

Datei öffnen

Öffnet die Datei, deren Definition als mit einem Null-Byte abgeschlossener Text vorliegt, auf den D1 zeigt.

Der Modus in D2 kann Mode_readwrite (1004 bei DOS 1.2) für Ein-/Ausgaben, Mode_old (1005) für Eingaben aus oder Mode_new (1006) für Ausgaben in die Datei sein.

In D0 wird ein Zeiger auf die Filehandle-Struktur zurückgegeben oder eine Null, wenn die Funktion nicht ausgeführt werden konnte. Die Filehandle-Struktur hat folgenden Aufbau:

Offset	Name	Bedeutung
0	Link	unbenutzt
4	Interact	wenn <>0, dann ist die Datei interaktiv
8	ID	Identifikationsnummer der Datei
12	Buffer	Zeiger auf intern benötigten Speicher
16	CharPos	intern benötigter Zeiger
20	BufEnd	intern benötigter Zeiger
24	ReadFunc	Zeiger auf Routine, die bei geleertem Puffer aufgerufen wird
28	WriteFunc	Zeiger auf die Routine, die bei gefülltem Puffer aufgerufen wird
32	CloseFunc	Zeiger auf die Routine, die beim Schließen der Datei aufgerufen wird
36	Argument1	
40	Argument2	Dateityp-abhängige Argumente

Die meisten Einträge sind nur für die interne Verwendung des AmigaDOS vorgesehen. Diese Werte sollten nicht manipuliert werden.

Close

```
Close ( Handle )
-36      D1
```

Datei schließen

Schließt die mit Open geöffnete Datei. Der in D1 übergebene Zeiger ist der von der Open-Funktion erhaltene Zeiger auf die Filehandle-Struktur.

Read

Anzahl = Read (Handle, Puffer, Länge)
 D0 -42 D1 D2 D3

Daten lesen

Liest aus der mit 'Handle' spezifizierten Datei bis zu 'Länge' Bytes in den Speicher ab Adresse 'Puffer'.

Der in D0 zurückgegebene Wert gibt die Anzahl der wirklich gelesenen Bytes an. Ist diese Zahl 0, so wurde das Ende der Datei erreicht. Tritt ein Fehler auf, so wird -1 zurückgegeben.

Write

Anzahl = Write (Handle, Puffer, Länge)
 D0 -48 D1 D2 D3

Daten schreiben

Schreibt in die mit 'Handle' spezifizierte Datei 'Länge' Bytes aus dem Speicher ab Adresse 'Puffer'.

In D0 wird die Anzahl der Bytes zurückgegeben, die wirklich geschrieben wurden. Ist dieser Wert -1, so ist ein Fehler aufgetreten.

Seek

Position = Seek (Handle, Abstand, Modus)
 D0 -66 D1 D2 D3

Dateizeiger verstellen

Diese Funktion verstellt den internen Zeiger in der mit 'Handle' spezifizierten Datei. Der 'Modus' bestimmt, ob der in 'Abstand' angegebene Wert den Zeiger relativ zum Dateianfang, zur aktuellen Position oder zum Dateende verstellen soll. Dieser Wert

wird von dort aus vorzeichenrichtig gerechnet, so daß auch rückwärts verschoben werden kann.

Die möglichen Modi sind:

OFFSET_BEGINNING	-1
OFFSET_CURRENT	0
OFFSET_END	1

Der Rückgabewert gibt die nach der Ausführung der Funktion aktuelle Position des Zeigers an. Um die momentane Position des Zeigers festzustellen, kann somit einfach der Modus 'relativ zur aktuellen Position' (OFFSET_CURRENT) eingestellt und um 0 Bytes verschoben werden: die zurückgegebene Position ist gleich der alten.

Input

```
Handle = Input ()  
D0      -54
```

Standard-Eingabekanal ermitteln

Diese Funktion ermittelt das Handle des Kanals, aus dem die standardmäßigen Eingaben gelesen werden können. Dies ist in dem Fall, wenn das Programm vom CLI aus aufgerufen wurde, das Handle des CLI-Fensters. Wird in dem CLI-Kommando, das das Programm aufgerufen hat, von der Möglichkeit der Datenumlenkung Gebrauch gemacht, so wird das Handle des gewählten Kanals ermittelt. Ein Beispiel:

```
>Programmname <DF0:Dateiname
```

läßt die mit Read erfolgenden Eingaben innerhalb des aufgerufenen Programmes aus der Datei 'Dateiname' erfolgen.

Output

```
Handle = Output()
```

Standard-Ausgabekanal ermitteln

Diese Funktion ermittelt das Handle des Kanals, in den die standardmäßigen Ausgaben geschrieben werden können. Dies ist in dem Fall, wenn das Programm vom CLI aus aufgerufen wurde, das Handle des CLI-Fensters. Auch hier kann die standardmäßige Ausgabe verstellt werden, wie z.B.

```
>Programmname >PRT:
```

die Standard-Ausgaben des aufgerufenen Programms zum Drucker schickt.

WaitForChar

```
Status = WaitForChar ( Handle, Timeout )  
D0      -204      D1      D2
```

Auf Empfang eines Zeichens warten

Diese Funktion wartet die in 'Timeout' angegebene Anzahl von Mikrosekunden auf den Empfang eines Zeichens aus dem mit 'Handle' spezifizierten Kanal (z.B. RAW:-Fenster, warten auf Tastendruck). Wird in dieser Zeit kein Zeichen empfangen, so wird in 'Status' eine 0 zurückgegeben, andernfalls der Wert -1. Das Zeichen kann dann mit der Read-Funktion ausgelesen werden.

Die Funktion ist nur verfügbar, wenn es sich bei dem Kanal um einen interaktiven Kanal handelt (virtuelles Terminal), z.B. ein RAW:-Fenster, in dem Ein- und Ausgaben gleichermaßen stattfinden können und die Daten nicht unbedingt sofort auf Anforderung kommen.

IsInteractive

```
Status = IsInteractive ( Handle )  
D0      -216          D1
```

Kanal-Typ ermitteln

In 'Status' wird TRUE (-1) zurückgegeben, wenn es sich bei dem mit 'Handle' spezifizierten Kanal um ein virtuelles Terminal handelt, mit dem Ein- und Ausgaben stattfinden können. Andernfalls erhält man FALSE (0) zurück.

IoErr

```
Error = IoErr ()  
D0      -132
```

Ein-/Ausgabefehler ermitteln

Wird nach dem Aufruf einer Funktion ein Fehler signalisiert, meist durch eine Null als Rückgabewert in D0, so kann durch den Aufruf von IoErr() die genaue Fehlermeldung ermittelt werden. D0 enthält dann die Nummer des zuvor aufgetretenen Fehlers. (vergl. WHY-Kommando des CLI)

Eine Auflistung der Fehlerwerte finden Sie im nächsten Kapitel.

2) Disketten-Operationen

CreateDir

```
Lock = CreateDir ( Name )
D0      -120      D1
```

Unterdirectory erstellen

Es wird das Unterdirectory 'Name' im aktuellen Directory erstellt.

Der Rückgabewert stellt einen Zeiger auf eine Datei-Struktur (Lock) dar, die folgenden Aufbau hat:

Offset	Name	Bedeutung
0	NextBlock	Zeiger auf nächsten mit diesem verketeten Lock oder Null
4	DiskBlock	Block-Nummer des Directories bzw. des File-Headers
8	AccessType	Zugriffs-Typ: -1= exklusiver Zugriff, -2= allgemeiner Zugriff
12	ProcessID	Identifikationsnummer
16	VolNode	Zeiger auf Disketten-Info

Diese Struktur stellt sozusagen den Schlüssel zu dieser Datei bzw. dem Unterdirectory dar, da mit ihr auf diese zugegriffen werden kann. (vergl. MAKEDIR-Kommando des CLI)

Lock

```
lock = Lock ( Name, Modus )
D0      -84      D1      D2
```

Dateischlüssel ermitteln

Es wird eine Datei oder ein Unterdirectory mit dem Namen 'Name' auf der Diskette gesucht und dafür eine Struktur erzeugt.

Der Modus bestimmt, welcher Art der Zugriff auf diese Datei sein soll. Ist er Lesen (-2), so kann aus dieser Datei von mehreren Tasks gelesen werden, ist er Schreiben (-1), so kann nur dieses Programm in die Datei schreiben.

CurrentDir

```
oldLock = CurrentDir ( Lock )  
D0      -126      D1
```

Unterdirectory zum aktuellen Directory erheben

Das durch 'Lock' spezifizierte Unterdirectory wird zum aktuellen Directory erhoben (siehe CD-Befehl des CLI).

Der zurückgegebene Wert stellt den Zeiger auf das vorher aktuelle Directory bzw. dessen Lock dar.

ParentDir

```
Lock_neu = ParentDir ( Lock )  
D0      -210      D1
```

Das übergeordnete Directory ermitteln

Das dem mit 'Lock' angegebenen Directory übergeordnete Directory wird ermittelt und dessen Lock in D0 zurückgegeben. Gehörte 'Lock' bereits zum obersten Directory (Root-Directory), so wird in D0 eine Null zurückgegeben.

DeleteFile

```
Status = DeleteFile ( Name )  
D0      -72      D1
```

Datei löschen

Die Datei mit dem angegebenen Namen wird gelöscht. Der Name muß als mit einem Null-Byte abgeschlossener Text vorliegen. In D0 wird eine Fehlermeldung zurückgegeben, wenn die Funktion nicht ausgeführt werden konnte (Datei nicht vorhanden, Datei schreibgeschützt, Directory nicht leer etc.).

Wird ein Unterdirectory zum Löschen angegeben, so dürfen in diesem keine Einträge mehr vorhanden sein.

Rename

```
Status = Rename ( Name_alt, Name_neu )  
D0      -78      D1      D2
```

Datei umbenennen

Die Datei oder das Directory mit dem in 'Name_alt' angegebenen Namen wird umbenannt. Sollte eine Datei mit dem neuen Namen bereits existieren, so wird abgebrochen und ein Fehler zurückgegeben.

Die beiden Namensangaben können auch eine Pfadangabe enthalten. In diesem Fall wird die Datei von dem alten in das neue Directory mit dem neuen Namen gebracht. Dies funktioniert allerdings nur auf ein und derselben Diskette.

DupLock

```
newLock = DupLock ( Lock )  
D0      -96      D1
```

Lock kopieren

Die alte Lock-Struktur wird in eine neue kopiert. D0 zeigt dann auf die neue Struktur. Dies kann verwendet werden, wenn mehrere Prozesse auf diese Datei zugreifen sollen. Es kann jedoch kein Lock kopiert werden, der nur zum Schreiben zugelassen ist, da dieser ohnehin nur für den exklusiven Zugriff zugelassen ist.

UnLock

```
UnLock ( Lock )  
-90      D1
```

Lock entfernen

Die Lock-Struktur, die mit Lock(), DupLock() oder CreateDir() erstellt wurde, wird entfernt und der belegte Speicher wieder freigegeben.

Examine

```
Status = Examine ( Lock, InfoBlock )  
D0      -102     D1  D2
```

Datei-Informationen holen

Die Struktur, auf die D2 zeigt, wird mit Informationen über die durch Lock spezifizierte Datei gefüllt. Diese Struktur wird FileInfoBlock genannt und ist folgendermaßen aufgebaut:

Offset	Name	Bedeutung
0	DiskKey.L	Diskettennummer
4	DirEntryType.L	Eintragstyp (+=Directory, -=Datei)
8	FileName	108 Bytes mit dem Dateinamen
116	Protection.L	Datei geschützt?
120	EntryType.L	Eintragstyp
124	Size.L	Dateilänge in Bytes
128	NumBlocks.L	Anzahl der damit belegten Blocks
132	Days.L	Erstellungsdatum
136	Minute.L	Erstellungszeit
140	Tick.L	Erstellungszeit
144	Comment	116 Bytes mit Kommentar

D0 enthält eine Null, wenn die Funktion nicht ausgeführt werden konnte.

ExNext

```
Status = ExNext ( Lock, InfoBlock )
D0      -108    D1    D2
```

Nächsten Directory-Eintrag ermitteln

Dieser Funktion wird der mit Examine() gefüllte InfoBlock sowie der Lock des gewählten Directories übergeben. Es werden nun die Informationen des ersten passenden Eintrages dieses Directories in den InfoBlock eingetragen. Bei einem weiteren Aufruf von ExNext() wird dann jeweils der nächste passende Eintrag dieses Directories gesucht und dessen Informationen zurückgegeben. Ist kein weiterer Eintrag mehr zu finden oder ein sonstiger Fehler aufgetreten, so wird in D0 eine Null zurückgegeben.

Mit den Befehlen Lock(), Examine() und ExNext() kann das Inhaltsverzeichnis einer Diskette ausgelesen werden. Der Weg ist folgender:

1. Mit Lock() wird der Schlüssel zu dem gewünschten Directory erstellt.

2. Mit Examine() kann nun der Directory-Name bzw. der Name der Diskette ermittelt werden. Gleichzeitig wird der FileInfoBlock erstellt, der für die nächste Funktion benötigt wird.
3. Durch mehrmaligen Aufruf der ExNext()-Funktion werden nun die einzelnen Einträge des Directories ausgelesen und deren Informationen im FileInfoBlock eingetragen. Dies wird sooft wiederholt, bis die ExNext()-Funktion eine Null zurückgibt: keine weiteren Einträge mehr vorhanden!

Hier ein kleines Maschinen-Programm, das diese Schritte ausführt. Die aufgerufene Print-Routine ist hier nicht aufgeführt, sie könnte z.B. den Namen und die Länge der momentan ausgelesenen Datei auf dem Bildschirm ausgeben.

Vor dem Aufruf dieser Routine muß die DOS-Bibliothek geöffnet werden und die DOS-Basisadresse in 'dosbase' abgelegt werden.

```

Lock      = -84
Examine   = -102
ExNext    = -108
IoErr     = -132

```

```

...
directory:                                ;* Inhaltsverzeichnis von DF0:
    move.l  dosbase,a6                    ;DOS-Basisadresse in A6
    move.l  #name,d1                     ;Zeiger auf Pfad-/Dateiname
    move.l  #-2,d2                        ;Modus 'Lesen'
    jsr     Lock(a6)                      ;Datei suchen
    tst.l   d0                            ;gefunden?
    beq     Error                         ;nein!
    move.l  d0,locksav                    ;sonst Schlüssel retten

    move.l  dosbase,a6                    ;DOS-Basisadresse
    move.l  locksav,d1                    ;Schlüssel in D1
    move.l  #fileinfo,d2                  ;Zeiger auf FileInfoBlock
    jsr     Examine(a6)                   ;Disk-Namen holen
    tst.l   d0                            ;OK?
    beq     error                         ;nein (kommt kaum vor)
    bra     ausgeben                      ;sonst Namen ausgeben

```

```

loop:                                ;* Dateinamen auslesen
    move.l dosbase,a6                ;DOS-Basisadresse
    move.l locksav,d1                ;Schlüssel in D1
    move.l #fileinfo,d2              ;Zeiger auf FileInfoBlock
    jsr ExNext(a6)                   ;nächste Datei suchen
    tst.l d0                          ;gefunden?
    beq error                        ;nein: Ende

ausgeben:                            ;* Namen ausgeben
    bsr Print                        ;Namen etc. ausgeben/auswerten
    bra loop                          ;und weitermachen...

error:                               ;* I/O-Status ermitteln
    move.l dosbase,a6                ;DOS-Basisadresse in A6
    jsr IoErr(a6)                    ;Status holen
    rts                              ;Ende...

name:    dc.b 'DF0:',0
even
locksav: blk.l 0
fileinfo: blk.l 260

```

Nach der Beendigung dieser Routine wird in D0 der Fehler-Code zurückgegeben, der durch die IoErr()-Funktion ermittelt wurde. Dieser Code sollte 232 (no_more_entries) sein, andernfalls ist irgendetwas schiefgegangen.

Info

```

Status = Info ( Lock, InfoData )
D0      -104  D1      D2

```

Disketten-Informationen holen

Der Parameter-Block, auf den D2 zeigt, wird mit Informationen über die verwendete Diskette gefüllt. Dieser Block muß an einer Adresse beginnen, die durch 4 teilbar ist (Longword aligned).

Lock muß zu der Diskette oder einer Datei bzw. einem Unterdirectory dieser Diskette passen.

Der Parameterblock InfoData hat folgenden Aufbau:

Offset	Name	Bedeutung
0	NumSoftErrors	Anzahl der Diskettenfehler
4	UnitNumber	installierte Disketten-Einheit
8	DiskState	Disketten-Status (s.u.)
12	NumBlocks	Anzahl der Blöcke auf der Diskette
16	NumBlocksUsed	Anzahl der verwendeten Blöcke
20	BytesPerBlock	Anzahl der Bytes pro Block
24	DiskType	Disketten-Typ (s.u.)
28	VolumeNode	Zeiger auf Disketten-Namen
32	InUse	<>0, wenn Diskette aktiv

DiskState zeigt den Status der Diskette an. Dabei gibt es folgende Möglichkeiten:

- 80 Diskette ist schreibgeschützt
- 81 Diskette wird gerade repariert (validating)
- 82 Diskette OK und beschreibbar

DiskType enthält den Typ der Diskette als Text, wenn eine eingelegt ist. Die möglichen Werte sind:

- 1 keine Diskette eingelegt
- BAD Diskette unlesbar (falsches Format)
- DOS DOS-Diskette
- NDOS Format stimmt, aber keine DOS-Diskette
- KICK Kickstart-Diskette

SetComment

```
Status = SetComment ( Name, Kommentar )
D0      -180      D1      D2
```

Datei-Kommentar setzen

Die Datei oder das Unterdirectory 'Name' wird mit einem Kommentar versehen. Der Kommentar darf bis zu 80 Zeichen lang sein und muß mit einem Nullbyte enden.

SetProtection

```
Status = SetProtection ( Name, Maske )
D0      -186          D1  D2
```

Dateistatus setzen

Der Schreib-/Lese-Status der angegebenen Datei bzw. des Unterdirectories wird gesetzt. Die unteren 4 Bits der Maske haben folgende Bedeutungen:

Bit	Bedeutung, wenn gesetzt
0	Datei nicht löschar
1	nicht ausführbar
2	nicht überschreibbar
3	nicht lesbar

3) Prozess-Verarbeitung**CreateProc**

```
Prozess = CreateProc ( Name, Pri, Segment, Stack )
D0      -138          D1  D2  D3  D4
```

Einen neuen Prozess erstellen

Es wird eine neue Prozess-Struktur unter dem Namen erstellt, auf den D1 zeigt. Dieser Prozess wird mit der in 'Pri' angegebenen Priorität laufen und einen Stack mit der Größe 'Stack' erhalten.

In 'Segment' wird ein Zeiger auf eine Segment-Liste übergeben (siehe auch LoadSeg), in der der zu startende Programmcode definiert wird. Im ersten Segment der Liste sollte dann das Programm beginnen.

Das Ergebnis der Funktion ist die neue Prozess-ID oder eine 0, wenn ein Fehler aufgetreten ist.

DateStamp

DateStamp (Vektor)
-192 D1

Datum und Uhrzeit ermitteln

In D1 wird ein Zeiger auf eine Tabelle aus drei Langworten zurückgegeben. Ist die Zeit im Amiga nicht gesetzt, so enthalten all diese Langworte eine 0. Andernfalls enthält das erste Langwort die Anzahl der vergangenen Tage seit dem 1. Januar 1978, das zweite die Anzahl der seit Mitternacht vergangenen Minuten und das dritte die in dieser Minute vergangenen 1/50 Sekunden. Dieser Wert ist jedoch immer ein Vielfaches von 50, so daß nur die Sekundenzahl*50 angegeben wird.

Delay

Delay (Zeit)
-198 D1

Den laufenden Prozeß kurzzeitig stoppen

Der laufende Prozeß wird um die in 'Zeit' angegebene Anzahl von 1/50 Sekunden angehalten.

DeviceProc

Prozeß = DeviceProc (Name)
D0 -174 D1

Den I/O verwendenden Prozeß ermitteln

Die Identifikation des Prozesses, der momentan den mit 'Name' angegebenen Ein-/Ausgabekanal verwendet, wird zurückgegeben oder eine 0, wenn kein Prozeß gefunden wurde.

Bezieht sich der Name auf einen auf Diskette liegenden Kanal, so kann mit der IoErr()-Funktion ein Zeiger auf die Lock-Struktur des entsprechenden Directories erhalten werden.

Exit

```
Exit ( Parameter )  
-144    D1
```

Programm beenden

Das laufende Programm wird beendet. War das Programm vom CLI aus aufgerufen worden, so wird diesem wieder die Kontrolle übergeben und der in 'Parameter' übergebene Integer-Wert als Rückgabewert interpretiert. Wurde das Programm als Prozeß gestartet, so wird durch Exit() dieser Prozeß gelöscht und der durch ihn verwendete Stack-, Segment- und Prozeß-Speicherbereich wieder freigegeben.

Execute

```
Status = Execute ( Kommando, Input, Output )  
D0      -222    D1      D2      D3
```

CLI-Kommando aufrufen

Das CLI-Kommando, das als Text vorliegt und auf das D1 zeigt, wird ausgeführt. Mit 'Input' und 'Output' können die Ein- und Ausgaben des CLI-Kommandos in irgendwelche Kanäle umgeleitet werden, deren Handle dann hier angegeben werden muß. Wird für Input oder Output eine 0 angegeben, so wird der Standard-Kanal verwendet.

Mit diesem Kommando können Sie sich leicht ein eigenes CLI erstellen, das z.B. ein eigenes Fenster öffnet und dann Execute() mit dem Fenster-Handle für Input und einem leeren Kommando-Text aufruft. Die Kommandos werden dann in dem Fenster eingegeben und die Ausgaben ebenfalls in dieses Fenster gebracht. Dieses eigene CLI kann auch mit dem ENDCLI-Befehl

beendet werden, wobei sich das Programm RUN im C:-Directory befinden muß.

LoadSeg

```
Segment = LoadSeg ( Name )  
D0      -150      D1
```

Programmdatei laden

Die Programmdatei 'Name' wird in den Speicher geladen. Das Programm wird eventuell über mehrere Speichermodule verteilt, wenn nicht genug zusammenhängender Speicherplatz verfügbar ist. Die dadurch entstehenden Segmente werden untereinander verkettet, indem der erste Eintrag jedes Segmentes ein Zeiger auf das nächste Segment der Liste ist. Ist dieser Zeiger 0, so ist dies das letzte Segment.

Tritt bei diesem Vorgang ein Fehler auf, so werden alle bereits geladenen Segmente wieder freigegeben und eine 0 in D0 zurückgegeben. Andernfalls enthält D0 einen Zeiger auf das erste Segment.

Das geladene Programm kann nun mit CreateProc() gestartet oder mit UnLoadSeg() wieder gelöscht werden.

UnLoadSeg

```
UnLoadSeg ( Segment )  
-156      D1
```

Geladene Programmdatei löschen

Die mit LoadSeg() geladene Programmdatei wird gelöscht und der verwendete Speicher wieder freigegeben. Der Zeiger in D1 weist auf das erste Segment der Liste (siehe LoadSeg).

GetPacket

```
Status = GetPacket ( Waitflag )  
D0      -162      D1
```

Paket abholen

Es wird ein Paket abgeholt, das von einem anderen Prozeß gesendet wurde. Ist das Waitflag TRUE (-1), so wird auf den Erhalt des Paketes gewartet, andernfalls wird nicht gewartet und eine Null zurückgegeben, wenn kein Paket vorliegt.

QueuePacket

```
Status = QueuePacket ( Paket )  
D0      -168      D1
```

Paket absenden

Das Paket, auf dessen Struktur D1 zeigt, wird abgeschickt. Ist dies einwandfrei geschehen, so wird in D0 ein Wert $\neq 0$ zurückgegeben.

3.1.4 DOS-Fehlermeldungen

In der folgenden Liste werden die mit IoErr() bzw. dem WHY-Kommando des CLI ermittelten Fehlercodes und deren Name bzw. Bedeutung aufgeführt.

103 insufficient free store

Es ist nicht genügend Speicherplatz frei.

104 task table full

Es sind bereits 20 Prozesse aktiv, mehr geht nicht.

120 *argument line invalid or too long*

Die Argumenteliste für diesen Befehl ist nicht korrekt oder enthält zu viele Angaben.

121 *file is not an object module*

Die aufgerufene Datei ist nicht lauffähig.

122 *invalid resident library during load*

Die aufgerufene residente Bibliothek ist ungültig.

202 *object in use*

Die angegebene Datei bzw. das Directory ist momentan von einem anderen Programm in Benutzung und nicht für andere Anwendungen verwendbar.

203 *object already exists*

Der angegebene Dateiname existiert bereits.

204 *directory not found*

Das gewählte Directory existiert nicht.

205 *object not found*

Der Kanal mit dem angegebenen Namen existiert nicht.

206 *invalid window*

Die Angabe der Parameter für das zu öffnende Fenster ist nicht korrekt.

209 *packet requested type unknown*

Die gewünschte Funktion ist auf dem angegebenen Gerät nicht möglich.

210 *invalid stream component name*

Der Dateiname ist ungültig (zu lang oder mit unerlaubten Zeichen versehen).

211 *invalid object lock*

Die angegebene Lock-Struktur ist ungültig.

212 *object not of required type*

Dateiname und Directoryname sind verwechselt worden.

213 *disk not validated*

Die Diskette ist entweder noch nicht vom System anerkannt oder defekt.

214 *disk write-protected*

Die Diskette ist schreibgeschützt.

215 *rename across devices attempted*

Die RENAME-Funktion ist nur innerhalb einer Diskette möglich.

216 *directory not empty*

Ein nicht leeres Directory sollte gelöscht werden.

218 *device not mounted*

Die gewählte Diskette ist nicht eingelegt.

219 *seek error*

Die Seek()-Funktion ist mit unerlaubten Parametern versehen.

220 *comment too big*

Der Kommentar zur Datei ist zu lang.

221 *disk full*

Die Diskette ist voll bzw. enthält nicht genug freien Platz für diese Anwendung.

222 *file is protected from deletion*

Die Datei ist nicht löschar bzw. gegen das Löschen geschützt worden.

223 *file is protected from writing*

Die Datei ist gegen Überschreiben geschützt.

224 *file is protected from reading*

Die Datei ist gegen Lesen geschützt. Bei den letzten drei Fehlermeldungen können Sie mit dem LIST-Befehl den Status der betroffenen Dateien überprüfen.

225 *not a DOS disk*

Diese Diskette ist nicht im AmigaDOS-Format formatiert.

226 *no disk in drive*

Im angegebenen Laufwerk ist keine Diskette eingelegt.

232 *no more entries in directory*

Die letzte ExNext()-Funktion konnte keinen passenden Eintrag in dem Directory finden.

3.2 Disketten

Der Amiga arbeitet sehr stark Diskettenorientiert, d.h. er lädt oft irgendetwas nach. Aus diesem Grunde ist es wichtig, daß die Informationen auf einer Diskette sicher untergebracht sind und einigermaßen schnell gefunden werden können. Wir wollen uns nun mit der Aufteilung der Disketten und der Interpretation der darauf liegenden Daten beschäftigen.

Die grundlegende Unterteilung einer Diskette sieht folgendermaßen aus:

- Seite bzw. Kopf-Nummer (0 oder 1)
- Spur oder Track oder Zylinder (0 bis 79)
- Sektor (0-10)

Jede Amiga-Diskette wird beidseitig bespielt, wodurch 2 Seiten verfügbar sind.

Jede Seite der Diskette ist ihrerseits in 80 Spuren unterteilt, die als konzentrische Ringe um ihren Mittelpunkt angeordnet sind. Die äußere Spur trägt die Nummer Null, die innere die Nummer 79. Diese Spuren werden auch als Tracks oder Zylinder bezeichnet.

Jede dieser Spuren wird wiederum in 11 Sektoren unterteilt, die auch von 0 an gezählt werden. Diese Sektoren werden auch als Blocks bezeichnet, wobei die Sektoren nur von 0 bis 10, die Blocks jedoch von 0 bis 1759 gezählt werden, da diese die logische Sektornummer der Diskette bezeichnet.

Jeder dieser Sektoren enthält 512 Bytes an verfügbaren Informationen, wodurch jede Diskette $512 * 11 * 80 * 2 = 901120$ Bytes tragen kann. Diese Anzahl ist allerdings nicht vollständig für Ihre Daten verfügbar, da die Verwaltung der Daten ebenfalls einigen Platz benötigt.

Amiga-Disketten-Aufteilung

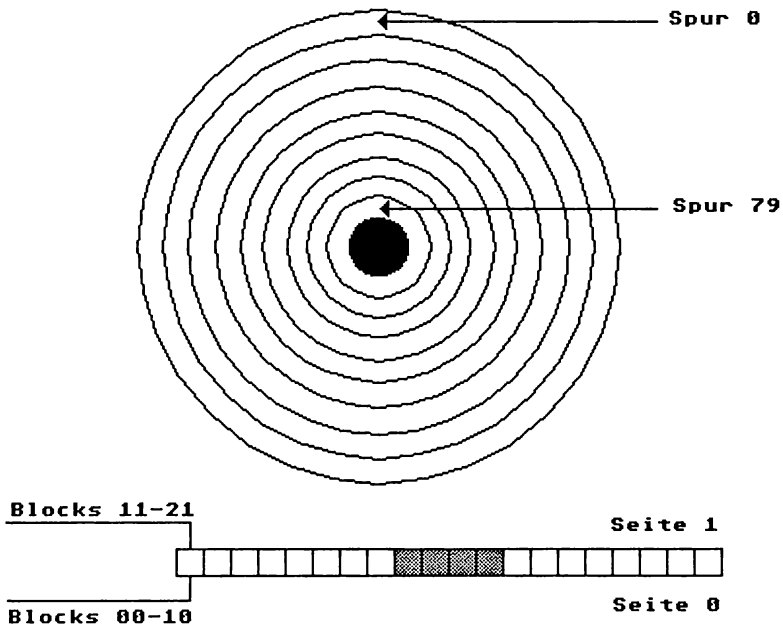


Abb. 3.1

Der erste logische Sektor, also der erste Block auf einer Diskette, liegt auf Seite 0, Spur 0, Sektor 0. Der darauffolgende Block ist der nächste Sektor dieser Spur und so weiter. Block 11 ist nun nicht der erste Sektor der zweiten Spur, sondern derjenige der ersten Spur auf der Rückseite (Seite 1) der Diskette. Auf diese Weise wird die Diskette immer abwechselnd oben und unten beschrieben bzw. gelesen.

3.2.1 Der Boot-Vorgang

Der erste Kontakt zur Diskette findet bereits statt, wenn man den Amiga einschaltet. Nachdem einige hardwaremäßige Initialisierungen des Rechners erledigt sind, läuft das Laufwerk 0 an. Was geschieht da?

Egal, ob in diesem Amiga das Kickstart bereits eingebaut ist oder nicht, es wird von der Diskette in diesem Laufwerk etwas geladen. Ist das Kickstart nicht eingebaut, so will der Rechner dies von der Diskette laden, andernfalls sucht er nach einer Workbench-Diskette. Dieser Ladevorgang beim Einschalten des Rechners wird 'booten' (sprich: buuten) genannt.

Das erste, was von der eingelegten Diskette geladen wird, sind die Boot-Blöcke, die die ersten beiden Sektoren (0 und 1) der Diskette belegen. Dort findet sich nun die Information darüber, um welchen Disketten-Typ es sich hier handelt. Die möglichen Typen sind:

- Kickstart
- DOS, eventuell ladbare DOS-Diskette (Workbench-Disk)
- unformatierte bzw. in einem fremden Format initialisierte Diskette

Die ersten vier Bytes des ersten Blocks der Diskette zeigen an, um welchen Typ es sich hier handelt. Dort stehen nämlich entweder die Buchstaben DOS mit abschließender Null für eine DOS-Diskette oder KICK für eine Kickstart-Diskette. Steht hier weder das eine noch das andere, so handelt es sich um eine fremde Diskette (BAD).

Die nun folgenden 4 Bytes stellen als Langwort die Checksumme des Boot-Blocks dar. Ist diese Summe korrekt, so nimmt der Amiga an, daß es sich hier um eine Workbench-Diskette handelt.

Das nächste Langwort enthält die Nummer des Diskettenblocks, der Root-Block genannt wird und normalerweise in Block \$370

(880) liegt. Die Bedeutung dieses Blocks folgt gleich, bleiben wir zunächst beim Boot-Block.

Beginnend im 7. Wort liegt hier nun ein Programm. Dieses Programm wird, wenn die Checksumme stimmt, als erstes gestartet. Es bekommt in A6 einen Zeiger auf die EXEC-Basisadresse übergeben und kann so direkt ein EXEC-Kommando ausführen.

Boot-Block (Sektor 0)

L-Wort	Name	Inhalt	Bedeutung
0	Disk-Type	DOS, KICK	Diskettentyp in 4 Buchstaben
1	Checksumme	???	Checksumme des Blocks
2	Rootblock	\$370	Blocknummer des Root-Blocks
3-127	Daten		Boot-Programm

Das Programm, das üblicherweise dort liegt, testet mittels des FindResident()-Kommandos des EXEC, ob die DOS-Bibliothek resident vorliegt. Ist dies nicht der Fall, so wird im Register D0 der Wert -1 übergeben. Wenn doch, so wird in D0 eine Null und in A0 ein Zeiger auf die Initialisierungs-Routine des DOS zurückgegeben.

Mit einem geeigneten Programm kann an dieser Stelle der gesamte Boot-Vorgang und somit die Initialisierung des Amiga selbst gestaltet werden. Sie haben hier also die Möglichkeit, eine eigene 'Workbench-Diskette' zu gestalten, da einige Disketten-Monitor-Programme die Erstellung der Checksumme übernehmen können. Diese Summe aller Worte des Blocks ist unbedingt nötig, damit der Amiga diesen Block als Boot-Block anerkennt.

3.2.2 Datei-Strukturen und Daten-Verteilung

Um auf einer Diskette mit einer Kapazität von ca. 800 KByte seine Daten so unterzubringen, daß sie später wieder auffindbar sind, gibt es bei der Verteilung von Daten einige Regeln. Diese Regeln sind dem AmigaDOS natürlich bekannt, so daß man sie eigentlich nicht unbedingt wissen mußte. Tritt jedoch einmal ein

Fehler auf einer Diskette auf, so muß man die verbliebenen Daten retten können.

Hierfür ist das DISKDOCTOR-Programm vorgesehen, das der Amiga bei einem Diskettenfehler sogar in einem Requester empfiehlt. Um die Arbeitsweise dieses Retters in der Not zu verstehen, sind eingehende Betrachtungen der Diskettenformate nötig.

Ein wichtiger Punkt hierbei ist die Verteilung der Dateien auf der Diskette und die Technik des Inhaltsverzeichnisses.

3.2.2.1 Diskettenaufteilung

Im Gegensatz zu den meisten Diskettenformaten ist das Inhaltsverzeichnis von Amiga-Disketten nicht in einigen zusammenhängenden Sektoren zu finden. Dies ist der Grund, weshalb die Ausgabe des Directory (z.B. mit dem CLI-Kommando DIR) so lange dauert.

Diese Methode hat Vor- und Nachteile. Der Nachteil ist die lange Zugriffszeit auf das Inhaltsverzeichnis, was teilweise schon ganz schön Nerven kostet. Dieser Nachteil wird jedoch durch einen großen Vorteil wettgemacht: die Möglichkeit der 'Reparatur' einer beschädigten Diskette.

Tritt bei einem der üblichen Disketten eines anderen Systems, z.B. des ATARI ST, ein Fehler ausgerechnet in Track 0 auf, wo das gesamte Inhaltsverzeichnis der Diskette liegt, so treten sehr große Probleme auf. Die Position der Daten und die Zusammenhänge der Sektoren sind nämlich dann nicht mehr bekannt und die Dateien somit nur mit einem Riesenaufwand wieder zu retten.

Dies ist beim Amiga nicht der Fall. Wie schon durch die Existenz des DISKDOCTOR-Programmes deutlich, sind die einzelnen Dateien relativ leicht wiederzufinden, ohne daß eine zentrale Verteilungsstelle notwendig ist. Dies wird durch große

Redundanzen erreicht, die zwar Speicherplatz auf den Disketten verbraucht, aber dadurch große Datensicherheit gewährleistet.

Wie funktioniert dies? Um die Struktur der Datenverteilung zu durchschauen, müssen wir nun den Aufbau der verschiedenen Diskettensektoren betrachten.

Root-Block

Abgesehen von den Bootsektoren befindet sich der Root-Block an einer festgelegten Stelle auf der Diskette. Dies ist üblicherweise auf Seite 0, Track 40, Sektor 0, und hat somit die Nummer 880 (\$370). Das dritte Langwort des Bootsektors enthält auch diese Nummer.

In diesem Block befindet sich die Wurzel der gesamten Diskette. Hier liegt das oberste Directory sowie der Diskettenname und dessen Erstellungs-Datum. Die Aufteilung dieses Blocks ist folgende (alle Werte sind Langworte, also 4 Bytes):

Root-Block

Wort	Name	Inhalt	Bedeutung
0	Type	2	Typ 2 (T.SHORT) bedeutet, daß es sich bei diesem Block um einen Anfangsblock einer Struktur handelt
1	Header Key	0	hat hier keine Bedeutung
2	High Seq	0	hat hier keine Bedeutung
3	HT-Size	\$48	dies ist die Größe der Tabelle, in der die Anfangsblocks der Files oder Underdirectories aufgeführt sind (Hashtable), die zusammen in einer Kette liegen
4	reserviert	0	hat hier keine Bedeutung
5	Checksumme	???	enthält einen Wert, der die Summe aller Worte dieses Blocks zu Null macht
6	Hashtable		hier beginnt die Tabelle, in der die Anfangsblocks der Dateien bzw. Underdirectories stehen

78	BM-Flag	-1	dieses Flag enthält -1 (TRUE), wenn die Bitmap der Diskette gültig ist
79	BM-Pages		die folgende Tabelle enthält Zeiger auf die Blocks, die die Bitmap enthalten. Meistens ist dies nur ein einzelner Block, so daß die übrigen Zeiger der Tabelle Null sind
105	Days		enthält das Datum, an dem die Diskette zuletzt verändert wurde
106	Mins		Uhrzeit der Änderung
107	Ticks		Sekunde der Änderung
108	Disk Name		hier liegt nun der Name der Diskette als BCPL-String, d.h. das erste Byte stellt die Anzahl der Zeichen des Namens (max. 30) dar
121	Create Days		Erstellungsdatum der Diskette
122	Create Mins		Erstellungszeit
123	Create Ticks		Erstellungssekunde
124	Next Hash	0	immer Null
125	Parent Dir	0	Zeiger auf übergeordnetes Directory: immer Null
126	Extension	0	immer Null
127	Sec. Type	1	dieses Wort stellt den Sekundär-Typ des Blocks dar. Für den Root-Block ist dies eine 1.

Die Werte in der Hashtable geben die Blocks an, in denen die Datei- bzw. Unterdirectory-Ketten innerhalb des Root-Directories beginnen. Da in diese Tabelle nicht genug Werte passen, werden aus den Dateien bzw. Unterdirectories Ketten gebildet, deren Namen einen bestimmten Zusammenhang haben.

Es wird aus dem Datei- bzw. Directorynamen ein Wert errechnet, der zwischen 6 und 77 liegt. Mit diesem Wert kann dann auf das Langwort der Hash-Tabelle zugegriffen werden, wo die gewählte Kette beginnt.

Die Funktion, nach der dieser Wert errechnet wird, ist folgende:

```
Hash=Länge des Namens
pro Buchstabe des Namens:
    Hash=Hash *13
    Hash=Hash +ASCII-Wert des Zeichens (immer Großbuchstaben)
    Hash=Hash & $7FF (logisch UND)
Hash=Hash modulo 72
Hash=Hash +6
```

Wie dies zu programmieren ist, finden Sie im Kapitel über die Trackdisk-Device, in dem ein Maschinenprogramm u.a. zur Auswertung der Hash-Tabelle vorgestellt wird.

Der Anfang der Kette liegt also dort, wohin der so berechnete Zeiger der Hash-Table zeigt. In dem so gefundenen Block steht dann im 124. Langwort die Nummer des nächsten Eintrages der Kette und so weiter, bis schließlich durch eine Null in diesem Zeiger das Ende der Kette angezeigt wird.

Diese Blocks, die den Anfang einer Datei- oder Directory-Struktur bilden, sind ebenfalls besonders aufgebaut. Beginnen wir mit dem ersten Block einer Datei, dem File-Header-Block.

File-Header-Block

Dieser Block enthält die Informationen über die entsprechende Datei. Darunter sind deren Name, Erstellungszeit und Kommentar sowie die Größe und Verteilung der Datei auf der Diskette. Die Aufteilung dieses Blocks ist folgende:

File-Header-Block

Wort	Name	Inhalt	Bedeutung
0	Type	2	Typ 2 (T.SHORT) bedeutet, daß es sich bei diesem Block um einen Anfangsblock einer Struktur handelt
1	Header Key		Hier steht die eigene Block-Nummer
2	High Seq		Enthält die gesamte Anzahl der Blocks dieser Datei
3	Data-Size	0	
4	first Data	0	Hier steht die Nummer des ersten Datenblocks der Datei. Dieser Wert findet sich auch in Wort Nr. 77 wieder
5	Checksumme	???	Enthält einen Wert, der die Summe aller Worte dieses Blocks zu Null macht
6	Data-Blocks		Hier beginnt die Tabelle, in der die Datenblocks der Datei aufgeführt sind. Die Tabelle beginnt bei Wort 77 und zählt dann rückwärts
78	reserviert	0	
79	reserviert	0	
80	Protect		Dieses Wort enthält in den unteren 4 Bits die Status-Information der Datei: Bit geschützt gegen, wenn gesetzt 0 löschen 1 verändern 2 überschreiben 3 lesen
81	Byte Size		Länge der Datei in Bytes
82	Comment		hier beginnt der Kommentar zu der Datei als BCPL-String (max. 22 Zeichen)
105	Days		enthält das Datum, an dem die Datei erstellt wurde
106	Mins		Uhrzeit der Erstellung
107	Ticks		Sekunde der Erstellung
108	Datei-Name		Hier liegt nun der Name der Datei als BCPL-String, d.h. das erste Byte stellt die Anzahl der Zeichen des Namens (max. 30) dar
124	Hash Chain		Blocknummer der nächsten Datei aus dieser Kette oder Null

125	Parent		Zeiger auf das Directory, in dem diese Datei aufgetaucht ist
126	Extension	0	Zeiger auf Erweiterungsblock oder Null
127	Sec. Type	-3	Dieses Wort stellt den Sekundär-Typ des Blocks dar. Für den File-Header-Block ist dies -3 (\$FFFD).

Eintrag Nr. 126 ist immer dann ungleich Null, wenn die Data-Block-Tabelle nicht lang genug ist, um alle Blocks dieser Datei aufzuzeigen. Ist dies der Fall, so wird dort auf einen Block gezeigt, der diese Liste fortführt.

File-List-Block

Dieser Block, der die Block-Liste fortführt, wird Erweiterungs-Block (File-List-Block) genannt und ist folgendermaßen aufgebaut:

File-List-Block

Wort	Name	Inhalt	Bedeutung
0	Type	\$10	Typ \$10 (T.LIST) bedeutet, daß es sich bei diesem Block um einen Erweiterungs-Block einer Datei-Struktur handelt
1	Header Key		Hier steht die eigene Block-Nummer
2	High Seq		enthält die gesamte Anzahl der Einträge in der Data-Block-Tabelle
3	Data-Size	0	
4	first Data	0	Hier steht die Nummer des ersten Datenblocks der Datei. Dieser Wert findet sich auch in Wort Nr. 77 des File-Header-Blocks wieder
5	Checksumme	???	Enthält einen Wert, der die Summe aller Worte dieses Blocks zu Null macht
6	Data-Blocks		Hier beginnt die Tabelle, in der die zusätzlichen Datenblocks der Datei aufgeführt sind. Die Tabelle beginnt bei Wort 77 und zählt dann rückwärts
78	info	0	Reserviert

124	Hash Chain	0	Blocknummer der nächsten Datei aus dieser Kette (immer Null)
125	Parent		Zeiger auf den File-Header-Block
126	Extension	0	Zeiger auf Erweiterungsblock oder Null
127	Sec. Type	-3	Dieses Wort stellt den Sekundär-Typ des Blocks dar

Die andere Möglichkeit eines Start-Blocks ist der User-Directory-Block, der am Anfang einer Directory-Struktur steht.

User-Directory-Block

Jedes Unterdirectory wird durch einen solchen Block begonnen, der ähnlich wie der Root-Block aufgebaut ist:

Wort	Name	Inhalt	Bedeutung
0	Type	2	Typ 2 (T.SHORT) bedeutet, daß es sich bei diesem Block um einen Anfangsblock einer Struktur handelt
1	Header Key		Hier steht die eigene Blocknummer
2	High Seq	0	Hat hier keine Bedeutung
3	HT-Size	0	Hat hier keine Bedeutung
4	reserviert	0	Hat hier keine Bedeutung
5	Checksumme	???	Enthält einen Wert, der die Summe aller Worte dieses Blocks zu Null macht
6	Hashtable		Hier beginnt die Tabelle, in der die Anfangsblocks der Dateien bzw. Unterdirectories stehen
78	reserviert	0	Hat hier keine Bedeutung
80	Protect		Dieses Wort enthält in den unteren 4 Bits die Status-Information der Datei:
			Bit geschützt gegen, wenn gesetzt 0 löschen 1 verändern 2 überschreiben 3 lesen
81	reserviert	0	Ohne Bedeutung

82	Comment		Hier beginnt der Kommentar zu diesem Underdirectory als BCPL-String (max. 22 Zeichen)
105	Days		Enthält das Datum, an dem dieses Underdirectory erstellt wurde
106	Mins		Uhrzeit der Erstellung
107	Ticks		Sekunde der Erstellung
108	Dir. Name		Hier liegt nun der Name des Directories als BCPL-String (max. 30)
124	Next Hash		Nächster Eintrag derselben Kette
125	Parent Dir		Zeiger auf das übergeordnete Directory
126	Extension	0	Immer Null
127	Sec. Type	2	Dieses Wort stellt den Sekundär-Typ des Blocks dar. Für den User-Directory-Block ist dies eine 2.

Außer diesen Struktur-Blocks sind natürlich auch Datenblocks auf der Diskette vorhanden. Diese haben den einfachsten Aufbau:

Data-Block

Wort	Name	Inhalt	Bedeutung
0	Type	8	Typ 8 (T.DATA) bedeutet, daß es sich hier um einen Datenblock handelt
1	Header Key		Hier steht die Blocknummer des File-Headers
2	Seq Num		Lfd. Nummer des Datenblocks in dieser Datei
3	Data-Size	\$1E8	Gültige Datenworte dieses Datenblocks (\$1E8 oder weniger)
4	Next Data		Nummer des nächsten Datenblocks dieser Datei
5	Checksumme	???	Enthält einen Wert, der die Summe aller Worte dieses Blocks zu Null macht
6	Data		Hier beginnen die Daten selbst

Nun fehlt nur noch der Block, der die beim Root-Block erwähnte Bitmap enthält. In diesem Block ist für jeden Block der Diskette ein Bit vorhanden, das anzeigt, ob jener Block

belegt oder frei ist. Der Aufbau des Bitmap-Blocks ist sehr einfach:

Bitmap-Block

Wort	Name	Inhalt	Bedeutung
0	Checksumme	???	Checksumme des Blocks
1-55	Bitmuster		Belegungs-Bits für alle Blöcke. Bit 0 des ersten Langworts steht für Block 2 usw. Ein gesetztes Bit bedeutet einen freien Block.

Nach dieser Betrachtung der Verteilung von Dateien über die Diskette wollen wir uns nun einmal ansehen, wie eine solche Datei überhaupt aufgebaut ist.

3.2.2.2 Programmaufbau

Eigentlich ist ein Programm nichts anderes als eine Reihe von binären Datenworten, die ein Maschinenprogramm bilden. Bei den 'guten, alten 8-Bitern' war daher die Abspeicherung eines solchen Programmes kein Problem: man mußte nur das Programm aus dem Speicher auf die Diskette schreiben und fertig.

Eine Maschine wie der Amiga wirft dabei allerdings schon eine Reihe von Problemen auf, die diese Methode unbrauchbar macht. Das erste Problem wäre die Aufteilung des Speichers. Ist der Speicher, wo das Programm zum ersten Mal gelaufen war, beim nächsten Aufruf bereits belegt, so wäre das einfache 'Reinladen' des Programmes nicht mehr so ohne weiteres möglich. Das Programm muß an eine andere Stelle des Speichers geladen werden, was für ein normales Maschinenprogramm, das absolute Adressen verwendet, bedeutet, daß es nicht mehr läuft.

Dieses Problem wird beim Amiga dadurch gelöst, daß ein Programm auf der Diskette so abgespeichert ist, daß alle absoluten Adressen im Programm für die Startadresse \$00000 ausgelegt sind. Wird das Programm nun z.B. ab \$20000 geladen, so müssen vor dem Start alle diese falschen Adressen korrigiert, d.h. um

\$20000 erhöht werden. Damit das DOS, das dies erledigt, die zu ändernden Adressen im Programm finden kann, ist zusätzlich zum eigentlichen Programm eine Tabelle abgespeichert. In dieser Tabelle stehen die Offsets, die jeweils auf auf ein zu änderndes Langwort zeigen.

Dies macht deutlich, daß ein einziger Abschnitt nicht ausreicht, ein Programm auf der Diskette abzuspeichern. Bisher sind es schon zwei Abschnitte, die wir kennen: das Programm selbst und die Tabelle, die Relocation-Tabelle genannt wird. Es gibt aber noch eine Reihe weiterer solcher Abschnitte, die der Amiga verwendet. Ein aus solchen Teilen zusammengesetztes Programmteil wird Hunk genannt. Ein oder mehrere Hunks ergeben zusammen eine Programm-Einheit (program unit), wovon eine oder mehrere ein Object-File bilden. Aus einem oder mehreren solcher Object-Files setzt sich schließlich ein Load-File zusammen, das ein lauffähiges Programm darstellt.

Der Unterschied zwischen diesen beiden File-Typen ist der, daß ein Object-File ein noch nicht lauffähiges Programm enthält, das z.B. von einem Compiler oder Assembler erstellt wurde. Will man ein oder mehrere dieser Files zu einem lauffähigen Programm machen, so muß man dafür den Linker aufrufen. Dies ist ein Programm, das Object-Files zu einem einzigen Programm zusammenbindet (engl. 'to link': verbinden), welches dann als Load-File abgespeichert wird. Das Ergebnis kann dann einfach durch die Eingabe seines Namens im CLI gestartet werden.

Der Vorteil dieses 'Umweges' ist der, daß so mehrere Teilprogramme, die sich ggf. gegenseitig aufrufen, einzeln erstellt und übersetzt werden können. Das Hauptprogramm, das z.B. in C geschrieben ist und die 'main'-Routine enthält, kann dann die in den anderen Dateien verteilten Funktionen oder Unterprogramme einfach aufrufen. Die Auslagerung der Funktionen aus dem Hauptprogramm verbessert somit die Übersichtlichkeit des Programmes, da es wesentlich kürzer als das gesamte Programm sein kann.

Der Grund dafür, daß die einzelnen Programm nicht alleine laufen können, wird somit klar. Es werden ja Programmteile

aufgerufen, die überhaupt nicht in diesem Programm enthalten sind! Erst nach dem Durchlauf des Linkers sind alle Funktionen und Unterprogramme in einem einzigen Programm-File enthalten.

Doch beginnen wir mit den kleinsten Abschnitten, aus denen sich die Hunks zusammensetzen. Einige dieser Programm-Datei-Teile kommen nur in Object-Files vor, einige nur in Load-Files. Sie beginnen jeweils mit einem bestimmten Langwort, das in der folgenden Tabelle in Klammern sedezimal mit angegeben wird.

Hier eine Übersicht über alle möglichen Hunk-Teile:

hunk_unit (\$3E7)

Mit diesem Teil beginnt eine Programm-Einheit in Object-Files. Nach der Kennung \$3E7 folgt die Länge des Namens dieser Einheit und danach der Name selbst, der an einer Langwort-Grenze enden muß.

hunk_name (\$3E8)

Hier liegt der Hunk-Name: nach der Kennung \$3E8 folgt die Namenslänge und der Name selbst, der an einer Langwortgrenze enden muß.

hunk_code (\$3E9)

Dieser Teil enthält ein Programmteil, das nach der Korrektur der absoluten Adressen laufen kann. Auch hier kommt nach der Kennung \$3E9 die Anzahl der Langworte des Programmes und dann die Langworte selbst.

hunk_data (\$3EA)

Auch hier beginnt ein Programmteil, allerdings nur Daten des Programms, die einen definierten Inhalt haben müssen (data). Einige dieser Daten können auch eine Adreß-Korrektur erfordern. Der Kennung folgt die Anzahl der Daten und die Daten selbst.

hunk_bss (\$3EB)

Die Daten dieses Teiles gehören zwar auch zum Programm selbst, haben aber keinen definierten Inhalt. Aus diesem Grund ist hier auch nach der Kennung nur die Anzahl der benötigten Langworte, nicht jedoch die Daten selbst aufgeführt.

hunk_reloc32 (\$3EC)

Dieser Block enthält die Offsets, die auf die zu korrigierenden Adreß-Langworte innerhalb des Programms zeigen. Diese Offsets sind für das gesamte Programm gültig. Die Aufteilung dieses Blocks ist folgende:

Nach der Kennung \$3EC folgt die Anzahl der Offsets, die in der ersten Tabelle enthalten sind. Das nächste Langwort bezeichnet die Nummer des Hanks, auf den sich diese Offsets beziehen, danach folgen die Offsets selbst. Das darauffolgende Langwort ist wieder eine Anzahl, danach folgt die Hunk-Nummer dieser Tabelle usw., bis als Anzahl eine Null auftritt und diesen Hunk-Teil beendet.

Das Ganze als Diagramm:

- \$3EC (hunk_reloc32)
 - Anzahl der Offsets
 - Hunk-Nummer
 - Offsets...
- Anzahl der Offsets (oder 0: Ende)
- Hunk-Nummer
- Offsets...
- ⋮
- 0: Ende von hunk_reloc32

Auf diese Weise kann diese Tabelle alle Hunks abdecken, aus dem das Programm, das schließlich im Speicher liegt und ablaufen soll, zusammengebaut ist.

hunk_reloc16 (\$3ED)

Diese Tabelle ist ebenso wie *hunk_reloc32* aufgebaut, nur daß diese Offsets sich auf 16-Bit-Adressen beziehen. Solche Adressen tauchen bei PC-relativen Adressierungen auf.

hunk_reloc8 (\$3EE)

Auch diese Tabelle hat das gleiche Format wie *hunk_reloc32*. Die hier enthaltenen Offsets werden bei 8-Bit-Adressen verwendet, die ebenfalls bei PC-relativen Adressierungen auftreten.

hunk_ext (\$3EF)

In diesem Block sind die Namen von externen Referenzen eingetragen. Solche Referenzen treten nur in Object-Files auf. Es handelt sich dabei um Adressen von Funktionen oder Unter-routinen, die dem Programmteil nicht bekannt sind und vom Linker eingesetzt werden müssen.

Der Kennung folgen mehrere sogenannte 'symbol data units', die durch ein Null-Wort abgeschlossen werden. Diese Symbol-Definitionen haben folgenden Aufbau:

1 Byte: Symbol-Typ. Hierfür gibt es folgende Möglichkeiten:

Name	Wert	Symbol-Typ
<i>ext_symb</i>	0	Symbol-Tabelle für Fehlersuche
<i>ext_def</i>	1	zu korrigierende Definition
<i>ext_abs</i>	2	absolute Definition
<i>ext_res</i>	3	Bezug auf residente Bibliothek
<i>ext_ref32</i>	129	32-Bit-Korrektur
<i>ext_common</i>	130	allgemeine 32-Bit-Korrektur
<i>ext_ref16</i>	131	16-Bit-Korrektur
<i>ext_ref8</i>	132	8-Bit-Korrektur

3-Byte-Wert für die Namenslänge (in Langworten)

Symbolname

Symbolwert und ggf. weitere Daten

hunk_symbol (\$3F0)

In diesem Block liegen ebenfalls Symbole mit ihrem Namen und Wert. Diese Symbole sind jedoch nicht für den Linker interessant, sondern für einen Debugger, ein Programm zur Fehlersuche in Programmen. In diesen Programmen kann dann eine Routine getestet werden, deren Adresse nicht als Zahl, sondern als Name verfügbar wird, ebenso wie die Speicherplätze von Variablen. Der Kennung \$3F0 folgen wieder symbol-data-units, abgeschlossen von einer Null.

hunk_debug (\$3F1)

Der Aufbau dieses Blocks ist nicht vollständig vorgeschrieben. Es können hier Informationen über das Programm eingetragen werden, über die das Programm zur Fehlersuche verfügen soll. Der Block muß nur mit der Kennung \$3F1 beginnen und die Anzahl der anschließenden Langworte folgen.

hunk_end (\$3F2)

Dies ist der einzige unbedingt nötige Block innerhalb des Hunks. Er besteht nur aus der Kennung, die somit auch das letzte Langwort eines Programmes auf der Diskette ist.

hunk_header (\$3F3)

Mit diesem Block beginnt ein Load-File. Hier wird angegeben, aus wievielen Hunks das zu ladende Programm besteht und wie groß diese jeweils sind. Außerdem enthält dieser Block die Namen der residenten Bibliotheken, die beim Laden dieses Programmes mitgeladen werden müssen. Der Aufbau ist folgender:

- `hunk_header ($3F3)`
 - Länge des Namens des ersten Hunks (in Langworten)
 - Hunkname
 - Länge des Namens des zweiten Hunks (oder 0: Ende)
 - Hunkname
 - ■
■
- 0: Ende der Namensliste
- höchste Hunk-Nummer+1: Tabellenlänge
- Nummer vom zuerst zu ladenden Hunk
- Nummer vom zuletzt zu ladenden Hunk
- hier beginnen die Hunks dieses Programmes

hunk_overlay (\$3F5)

Dieser Block wird benötigt, wenn mit Overlay gearbeitet werden soll. Dies bedeutet, daß in einen bereits vom Programm belegten Speicherbereich ein anderes Programm- bzw. Datensegment geladen werden soll. Die Tabelle nach der Kennung \$3F5 enthält die Angaben über die Tabellengröße, die höchste Ebene der Überlagerungen (die Anzahl der Überschreibvorgänge) und die nachzuladenden Daten selbst.

hunk_break (\$3F6)

Mit dieser alleine stehenden Kennung wird das Ende eines Overlay-Programmteils gekennzeichnet.

Um diese Aufteilung von Programmen etwas zu entwirren und zu demonstrieren, hier ein kleines Beispiel. Im CLI-Kapitel habe ich Ihnen ein kleines Maschinen-Programm vorgestellt, das den CLI-Befehl 'FONT' darstellt. Wie dieses Programm (vom K-SEKA-Assembler) auf die Diskette gebracht wird, können Sie sich leicht ansehen, indem Sie mit dem TYPE-Kommando des

CLI den Inhalt der Programmdatei 'Font' ausgeben lassen. Das können Sie mit dem Kommando

```
>type Font opt h
```

auf dem Bildschirm oder mit

```
>type Font to PRT: opt h
```

auf dem Drucker bewirken. Sie erhalten dann folgenden Ausdruck:

```
0000: 000003F3 00000000 00000002 00000000
0010: 00000001 00000024 00000001 / 000003E9
0020: 00000024 / 53406700 00180C18 00206600
0030: 000A51C8 FFF66000 000813E0 0000007F
0040: 2C790000 000443F9 00000072 70004EAE
0050: FE6823C0 00000088 67000028 2C790000
0060: 00884EAE FFC42200 243C0000 007E263C
0070: 00000009 2C790000 00884EAE FFD06000
0080: 0008203C FFFFFFFF 22002C79 00000088
0090: 4EAEFF70 4E75646F 732E6C69 62726172
00A0: 79009B30 3B33313B 34306D00 00000000
00B0: 3B4F7065 / 000003EC 00000007 00000000
00C0: 00000018 00000024 00000030 0000003A
00D0: 00000046 00000052 00000068 00000000 /
00E0: 000003F2 / 000003EB 00000001 / 000003F2
```

Die Schrägstriche erhalten Sie nicht bei der Ausgabe, diese sollen nur jetzt die einzelnen Abschnitte bzw. Hunk-Teile trennen. Sehen wir uns diese Teile einmal an:

Zu Beginn steht die Kennung \$3F3, hunk_header. Die darauffolgende \$0 bedeutet, daß keine Hunk-Namen vorliegen. Danach gibt die \$2 an, daß dieses Programmfile aus nur zwei Hunks besteht (wie gesagt, nur ein einfaches kleines Beispiel). Das erste zu ladende Hunk ist die Nummer \$0, das letzte die Nummer \$1. Die Größen dieser beiden Hunks sind \$24 und \$1.

Mit der Kennung \$3E9 (hunk_code) beginnt nun der Teil, in dem die Programmdaten selbst stehen. Die Länge wird mit \$24

angegeben. Danach folgen \$24 (36) Langworte des Programmcodes.

Danach beginnt mit der Kennung \$3EC (`hunk_reloc32`) der Bereich mit den Offsets für die Korrektur der Adressen. Die Anzahl wird mit \$7 angegeben, und diese Offsets beziehen sich auf Hunk Nummer \$0. Nun folgen die 7 Offsets selbst. Der erste dieser Offsets, \$18, deutet auf den Wert \$0000007F, das \$18. Wort des Codes. Zu diesem Langwort wird also nach dem Laden des Programmes die Anfangsadresse addiert, so daß dort die effektive Speicheradresse des adressierten Bytes hinkommt. Ebenso wird mit den anderen Offsets der `reloc32`-Liste verfahren.

Der Liste folgt nun eine \$0, was das Ende der `hunk_reloc32`-Liste bedeutet.

Die darauffolgende Kennung \$3F2 (`hunk_end`) gibt nun das Ende des ersten Hunks an. Es folgt nun also das zweite, dessen Länge ja Eins war.

Nun folgt die Kennung \$3F2 (`hunk_bss`), der die Zahl \$1 folgt. Dies bedeutet, daß ein Langwort zusätzlich reserviert werden muß, in das das Programm ja die DOS-Basisadresse legen will.

Den Abschluß bildet nun noch die Kennung \$3F2 (`hunk_end`), was das Ende des zweiten Hunks und auch des gesamten Programmes darstellt.

3.2.2.3 Das IFF-Format

IFF steht für Interchange File Format, so daß IFF-Format eigentlich doppelt gemoppelt ist. Es handelt sich dabei um ein Format, in dem diverse Datenfiles aufgebaut werden, damit sie von jedem Programm gelesen und ausgewertet werden können.

Eigentlich gehört dies gar nicht in ein Amiga-Intern-Buch, da das IFF nicht unmittelbar mit dem Amiga zu tun hat. Es wurde vielmehr von der Firma Electronic Arts entworfen und hat sich

so sehr zum Standard entwickelt, daß es inzwischen überall auftaucht. Aus diesem Grund soll hier ein Überblick über die Struktur des IFF gegeben werden.

Wofür braucht man so ein standardisiertes Format? Stellen Sie sich dafür ein Bild vor, das mit irgendeinem Programm auf dem Amiga gemalt wurde. Dieses Bild besteht aus einer Anzahl Daten, die auf dem Bildschirm die verschiedenen Farben des Bildes erscheinen lassen.

Wenn Sie diese Daten nun einfach auf die Diskette bringen, so wird es später beim Laden schon Probleme geben: Wie groß ist denn das Bild, auf dem diese Daten verteilt werden sollen? Welche Farbmischungen sollen verwendet werden?

Wie Sie sehen, mit den Bilddaten selbst ist es nicht getan. Sie müssen außerdem noch einige andere Informationen in der Datei unterbringen, und zwar so, daß ein anderes Programm sie auch wiederfinden kann.

Dieses Problem wurde durch die Einführung des IFF gelöst. Hier werden die Daten in einer fest definierten Art und Weise aufgeteilt und gespeichert. Jeder der verschiedenen Datenblöcke erhält dafür einen Header, einen bestimmten Vorspann, der aus einem Wort von 4 Buchstaben und einem Datenwort mit der Blocklänge besteht.

Den Anfang einer IFF-Datei bildet das Wort FORM, was bedeutet, daß hier ein Datenbereich einer bestimmten Anwenderform (Text, Bild etc.) beginnt. Das nun folgende Langwort gibt die Länge der Form an, die hier beginnt. Diese Form ist eine Kombination aus einigen Datenblöcken, die Chunks genannt werden. Eine IFF-Datei kann theoretisch auch aus mehreren Forms bestehen, wenn z.B. eine Text- und eine Bild-datei kombiniert wurde. Üblicherweise besteht eine Datei jedoch aus nur einer Form, da es sich ja meist nur um eine Text-, Bild- oder Sound-Datei handelt.

Nach dem Wort FORM folgt also ein Langwort mit der Länge dieser Form in Bytes, was üblicherweise der Dateilänge-8 ent-

spricht. Danach kommt ein weiteres 4-buchstabiges Wort, das den Typ dieser Datei angibt (ILBM, WORD etc.).

Unmittelbar auf den Typ folgt nun die Kennung des ersten Chunks, gefolgt von dessen Länge. Nach dieser Anzahl Daten folgt, eventuell mit einem Füllbyte auf eine gerade Adresse gebracht, der nächste Chunk usw., bis das Ende der Form und damit meist das Ende der Datei erreicht ist.

Das ganze im Überblick:

'FORM'	;Beginn einer IFF-Form
Formlänge	;Länge der Form in Bytes
Typ	;Kennung, z.B. ILBH, WORD, SMUS, 8SVX
Chunkname	;Name des Chunks, z.B. NAME, AUTH, BODY
Chunklänge	;Länge des Chunks in Bytes
usw...	

Es gibt inzwischen eine sehr große Anzahl von möglichen Chunk-Arten. Hier eine Übersicht über die wichtigsten Arten mit Kennung und Bedeutung:

Dateityp: Text-Datei (WORD)

BODY	Haupt-Datenteil für Bilder
COLR	Farben des Textes
DOC	Textart
FOOT	Fußzeile
FONT	verwendete Zeichensätze
FSCC	Text-Farb-Information
HEAD	Kopfzeile
PARA	Layout-Informationen (linker und rechter Rand etc.)
PCTS	Informationen über in den Text zu integrierende Bilder

PINF	Informationen über die Bilder selbst
TABS	Tabulator-Informationen
TEXT	eigentlicher Text-Teil

Dateityp: Graphik-Datei (ILBM)

BMHD	Graphik-Steuerdaten
CMAP	Farbtabelle
BODY	Graphik-Daten

Dateityp: Musik-Datei (SMUS)

SHDR	Sound-Steuerdaten (Tempo, Lautstärke, Tonkanal)
NAME	Name des Musikstückes
(c)	Copyright-Vermerk
AUTH	Name des Autors
ANNO	Bemerkungen zu diesem Stück
TRAK	Kanal-Angabe

Dateityp: 8-Bit-digitalisierter-Sound-Datei (8SVX)

VHDR	Steuerdaten (Typ, Tempo, Oktave, Lautstärke)
NAME	Name des Klages
(c)	Copyright-Vermerk
AUTH	Name des Autors
ANNO	Bemerkungen zu diesem Sound
BODY	Sound-Daten
ATAK	Attack-Informationen
RLSE	Release-Informationen

Wenn Sie eine IFF-Datei auf einer Diskette vorliegen haben, so können Sie sich deren Aufteilung einmal ansehen. Dafür folgt nun ein kleines Maschinenprogramm, das alle Kennungen mit deren Längen in einem Fenster ausgibt. Der Name der Datei wird dabei direkt in das Programm eingesetzt.

;***** IFF-Demo-Programm 6/87 S.D. *****

OpenLib = -408
 closelib = -414
 ExecBase = 4

Open = -30
 Close = -36
 Seek = -66
 Read = -42
 Write = -48
 mode_old = 1005

key = \$bfec01 ; Sondertasten-Status

run:

```

move.l execbase,a6 ;Zeiger auf EXEC-Bibliothek
lea dosname(pc),a1
moveq #0,d0
jsr openlib(a6) ;DOS-Library öffnen
move.l d0,dosbase
beq error

```

```

move.l #consolname,d1 ;Consol-Definition
move.l #mode_old,d2
move.l dosbase,a6
jsr open(a6) ;CON:-Fenster öffnen
beq error
move.l d0,conhandle

```

```

move.l #filename,d1
move.l #mode_old,d2
move.l dosbase,a6
jsr open(a6) ;File öffnen
beq error
move.l d0,filehandle

```

loop:

```

cmp.b #$37,key ;Alternate gedrückt?
beq qu ;ja: Abbruch

```

```

move.l #-1,d0

```

del:

```

dbra d0,del ;kleine Pause für's Ablesen...

```

```
bsr    read4            ;Deklarator einlesen
beq    qu               ;EOF
bmi    error            ;Error

move.l conhandle,d1
move.l #buffer,d2       ;Buffer-Adresse
move.l #6,d3            ;4 Zeichen
jsr    write(a6)         ;Deklarator ausgeben
beq    error

move.l buffer,d5        ;Deklarator retten
cmp.l  #'FORM',d5       ;FORM?
bne    noform           ;nein
st     flag             ;sonst Flag setzen
bra    form             ;und weiter

noform:
tst    flag             ;Kennung?
beq    form             ;nein
clr    flag             ;sonst Flag löschen
move.l #'----',outpuff  ;kennzeichnen
bsr    print
bra    loop             ;und weiter

form:
bsr    read4            ;Länge einlesen
beq    qu               ;EOF
bmi    error            ;Error
move.l buffer,d0        ;Wert in D0
bsr    phex             ;und ausgeben

cmp.l  #'FORM',d5       ;FORM?
beq    loop             ;ja: nächster

move.l filehandle,d1
move.l buffer,d2
addq.l #1,d2
bclr   #0,d2            ;auf gerade Adresse
move.l #0,d3            ;Modus: OFFSET_CURRENT
jsr    Seek(a6)         ;nächsten Teil suchen
bra    loop             ;weiter...

qu:
move.l conhandle,d1
move.l #endtext,d2      ;Ende-Text
move.l #25,d3
```

```

    jsr    Write(a6)        ;ausgeben

    move.l conhandle,d1
    move.l #buffer,d2      ;Buffer-Adresse
    move.l #1,d3           ;1 Zeichen
    jsr    Read(a6)        ;einlesen
    bra    ende

error:
ende:
    move.l conhandle,d1    ;Fenster schließen
    move.l dosbase,a6
    jsr    Close(a6)

    move.l filehandle,d1   ;Datei schließen
    jsr    Close(a6)

    move.l dosbase,a1      ;DOS.Lib schließen
    move.l execbase,a6
    jsr    CloseLib(a6)

    rts                   ;Ende!

read4:                      ;4 Zeichen einlesen
    move.l filehandle,d1
    move.l #buffer,d2      ;Buffer-Adresse
    move.l #4,d3           ;4 Zeichen einlesen
    jmp    Read(a6)

phex:                      ;D0 sedezimal ausgeben
    lea    outpuff,a0
    move    d0,d2
    move    #3,d3          ;4 Ziffern
niblop:
    rol    #4,d2           ;linkes Nibble nach unten
    move    d2,d1
    and    #$f,d1         ;ausmaskieren
    add    #$30,d1        ;zu ASCII machen
    cmp    #'9',d1        ;Ziffer?
    bls    nibok          ;ja
    add    #7,d1          ;sonst korrigieren
nibok:
    move.b d1,(a0)+        ;Zeichen in Ausgabepuffer
    dbra   d3,niblop       ;Schleife fortführen
    move.b #$a,(a0)       ;Return ans Ende

```



```
print:
    move.l dosbase,a6
    move.l conhandle,d1
    move.l #outpuff,d2      ;Ausgabepuffer
    move.l #5,d3            ;5 Zeichen
    jmp     Write(a6)       ;ausgeben

dosbase:    dc.l 0
conhandle:  dc.l 0
filehandle: dc.l 0
flag:       dc.w 0
outpuff:     dc.b '      '
buffer:      dc.b '      '
consolname:  dc.b 'RAW:0/10/400/240/** IFF-Format',0
dosname:     dc.b 'dos.library',0
filename:    dc.b 'IFF-Datei',0
endtext:     dc.b '** Bitte Taste drücken **'
even
```

Das Programm öffnet ein Fenster und gibt darin die Kennungen und Längen der einzelnen Chunks innerhalb der Datei an. Der Name der Datei muß bei 'filename:' in das Programm eingetragen werden. Bei der Ausgabe wurde zusätzlich eine kleine Warteschleife eingebaut, damit die Ausgaben leichter abzulesen sind. Wenn es Ihnen allerdings zu lange dauert oder die Datei gar keine IFF-Datei sein sollte, so können Sie durch Drücken der Alternate-Taste das Programm beenden. Es wird dann nur noch auf einen beliebigen Tastendruck gewartet und das Fenster geschlossen.

3.3 Programme

Ein Programm, welches von einem Linker oder direkt von einem Assembler erstellt wurde, kann einfach durch die Eingabe seines Namens im CLI gestartet werden. Will man es von der Workbench aus starten, so muß man zusätzlich noch eine .info-Datei erstellen, die das Bild des Programmes im Workbenchfenster enthält. Dieses Bild kann dann angeklickt werden, und das Programm wird gestartet.

3.3.1 Programmstart und Parameter

Wie man bereits von den CLI-Kommandos her weiß, gibt es beim Amiga die Möglichkeit, in der das Programm aufrufenden Zeile einige Parameter mit anzugeben, die das Programm dann übernehmen und auswerten kann. Eine solche Zeile kann beim Start aus der Workbench natürlich nicht so übergeben werden. Aus diesem Grund gibt es einen deutlichen Unterschied in der Parameterübergabe an das Programm zwischen dem CLI und der Workbench.

Das Programm, das aufgerufen wird, muß deshalb feststellen, von welcher Oberfläche aus es gestartet wurde, und dann eventuell seine Parameter auf die entsprechende Art und Weise abholen. Betrachten wir dafür zuerst einmal den einfacheren Fall, und zwar den Start eines Programms mittels des CLI.

3.3.1.1 Aufruf mit CLI

Das Programm, das vom CLI aus gestartet wurde, bekommt in zwei Registern die nötigen Informationen über die Parameter, die ggf. dem Namen folgend eingegeben wurden. Im Adreßregister A0 wird die Adresse der Zeile im Speicher übergeben, wo der dem Programmnamen folgende Text der im CLI eingegebenen Zeile liegt. Zusätzlich wird im Datenregister D0 die Anzahl der Zeichen mitgeteilt, die hinter dem eigentlichen Programmnamen liegen.

Mit diesen beiden Informationen kann nun das Programm leicht die Parameter auslesen und auswerten. Um dies einmal zu demonstrieren, folgt nun ein kleines Maschinenprogramm, das mit und ohne Parameter aufrufbar ist.

Es handelt sich bei diesem Programm um ein CLI-Kommando, das Sie sich auch in den C-Ordner kopieren können. Es hat die Aufgabe, die Darstellungsart der auf den Aufruf folgenden Texte zu bestimmen. Sie können dies z.B. in der Startup.Sequence verwenden, wenn Sie eine Meldung etwa unterstrichen oder kursiv ausgeben wollen.

Wenn Sie das Programm im C-Ordner unter dem Namen 'Font' liegen haben, können Sie es mit dem Kommando

>Font n

aufrufen. Der Parameter n kann dabei auch weggelassen werden, das Programm schaltet dann wieder die normale Textdarstellung ein.

Wenn Sie ihn dennoch angeben, so muß er eine Ziffer zwischen 0 und 7 sein. Die Wirkungen dieser Ziffern sind:

0	Normal-Darstellung
1	Fettschrift
3	kursive (schräge) Schriftart
4	unterstrichen
7	inverse Schrift

Sie können auch fett und unterstrichen einstellen, wenn Sie Font 1 und Font 4 hintereinander aufrufen.

Hier nun das Programm:

```
;***** FONT-Befehl *****  
  
; EXEC-Offsets  
  
OpenLib = -30-378  
ExecBase = 4  
  
; AmigaDOS Offsets  
  
Write    = -30-18  
Output   = -30-30  
Exit     = -30-114  
  
run:  
    subq    #1,d0            ;Byte-Anzahl-1  
    beq     normal          ;keine Parameter?  
search:  
    cmp.b   #$20,(a0)+      ;Argument suchen  
    bne     found           ;gefunden  
    dbra    d0,search
```

```

        bra        normal        ;normalen Font setzen

found:
        move.b     -(a0),text+1   ;Stil setzen

normal:
        move.l     execbase,a6     ;Zeiger auf EXEC-Bibliothek
        lea        dosname,a1
        moveq      #0,d0
        jsr        OpenLib(a6)    ;Open DOS-Library
        move.l     d0,dosbase
        beq        error          ;nicht geklappt

        move.l     dosbase,a6
        jsr        Output(a6)     ;Standard-Output-Handle holen

        move.l     d0,d1          ;Output-Handle in D1
        move.l     #text,d2       ;Text-Adresse in D2
        move.l     #tende-text,d3 ;Länge in D3
        move.l     dosbase,a6
        jsr        Write(a6)      ;Text ausgeben
        bra        ende           ;OK: Ende

error:
        move.l     #1,d0          ;Error-Status
ende:
        move.l     d0,d1          ;Rückgabe-Parameter
        move.l     dosbase,a6
        jsr        exit(a6)       ;Ende des Programmes

        rts                     ;kommt eigentlich nicht wieder

dosbase: dc.l 0

dosname: dc.b 'dos.library',0
text:    dc.b $9b,'0;31;40m'
tende:

even

```

Wenn Sie nun ein C-Programm schreiben, das diese Parameterzeile benötigt, so können Sie dies auch leicht erreichen. Sie müssen nur die Datei 'startup.o' als erstes Element in der Linker-Anweisung einsetzen, was auch normalerweise immer

getan wird. Die Parameterzeile finden Sie dann in der Variablen 'argv', die Anzahl der Zeichen in 'argc'.

Das Startup-Programmteil erledigt noch mehr. Es öffnet auch die DOS-Bibliothek und stellt mit den DOS-Funktionen Input() und Output() den Standard-Ein- und -Ausgabekanal fest. Die Handles dieser Kanäle finden Sie dann in 'stdin' und 'stdout'. Die Routine startet danach die 'main'-Routine ihres C-Programmes.

Eine weitere Information, die vom CLI an das Programm übergeben wird, ist die Größe des reservierten Stack-Bereiches. Dieser liegt hinter der Rücksprungadresse zum CLI auf dem Stack und kann z.B. mit dem Befehl

```
MOVE.L 4(SP),D0
```

eingelezen werden. Auf diese Weise kann das Programm testen, ob es für seine speziellen Anforderungen genügend Platz auf dem Stack hat.

Zusätzlich zu diesen Parametern werden noch einige andere vom CLI aus übergeben. Diese Parameter bieten viele Möglichkeiten, ein CLI-Programm zu vereinfachen. Näheres dazu finden Sie in dem Kapitel über die transienten CLI-Kommandos.

Dies ist also der Vorgang, ein vom CLI aus gestartetes Programm zu initialisieren. Betrachten wir nun den anderen Fall: den Start aus der Workbench.

3.3.1.2 Start von der Workbench aus

Wenn Sie das Icon eines Programmes, das als Bild in einem Workbench-Fenster vorliegt, mit einem doppelten Klick starten, so wird das Programm unter dem angezeigten Namen gestartet. Diesem Programm werden dann ebenfalls Parameter übergeben, allerdings nicht in Form einer Textzeile, sondern als Message.

Haben Sie dieses Programm in C geschrieben und ihm mit dem Linker das Startup-Programm vorangesetzt, so brauchen Sie sich um diese Message, die sogenannte Startup-Message, nicht zu kümmern. Das Startup-Programm erledigt selbständig folgende Arbeiten, wenn es festgestellt hat, daß es von der Workbench aus gestartet wurde:

1. Es öffnet zuerst einmal die DOS-Bibliothek.
2. Nun wird auf die Startup-Message gewartet (WaitPort).
3. Die Message wird abgeholt (GetMsg).
4. Die Anzahl der Argumente innerhalb der Message wird getestet. Ist sie 0, so wird der nächste Schritt übersprungen.
5. Die Argumente, die übergeben wurden, werden als Lock-Struktur interpretiert und mit ihr das dazugehörige Directory zum aktuellen Directory erklärt (CurrentDir).
6. Das Argument `sm_ToolWindow` wird überprüft. Ist es nicht 0, so wird das angegebene Fenster geöffnet und dessen Handle, wenn es sich öffnen ließ, zur Standard-Eingabe erklärt.

Wie muß nun ein Programm aussehen, das nicht über dieses komfortable Startup-Programm verfügt, etwa ein Maschinenprogramm?

Wenn Sie die Message, die die Workbench ihrem Programm sendet, nicht benötigen, so müssen Sie sie trotzdem abholen. Andernfalls wird der Guru wieder einmal meditieren, da bei der nächsten I/O-Funktion, etwa beim Öffnen eines Fensters, eine Meldung im Message-Port ankommen wird, die nicht zu dieser Funktion passt.

Sie müssen also in Ihrem Programm die gleichen Funktionen ausführen wie das Startup-Programm. Zuerst rufen Sie die `FindTask()`-Funktion des EXEC auf, um einen Zeiger auf die

Struktur des Prozesses, also Ihres Programmes, zu bekommen. Als Argument übergeben Sie dafür eine Null in A1:

```
execbase = 4
FindTask = -294
WaitPort = -384
GetMsg    = -372
```

```
move.l  execbase,a6      ;EXEC-Basisadresse in A6

suba.l  a1,a1            ;Argument A1 löschen
jsr     FindTask(a6)     ;Zeiger holen
```

In D0 erhalten Sie den Zeiger auf Ihre Prozeß-Struktur. In dieser Struktur finden Sie nun die Information, ob dieser Prozeß aus dem CLI oder von der Workbench aus gestartet wurde:

```
move.l  d0,a4            ;Zeiger auf Prozeß in A4
tst.l   $ac(a4)          ;pr_CLI: CLI oder Workbench?
bne     fromCLI          ;es war CLI!
```

Ist das getestete Argument Null, so wurde das Programm von der Workbench aus gestartet.

Ist dies der Fall, muß als nächstes auf den Empfang der Startup-Message gewartet werden. Dies wird mit der WaitPort()-Funktion realisiert:

```
lea     $5c(a4),a0       ;pr_MsgPort: MessagePort in A0
jsr     WaitPort(a6)     ;auf Message warten
```

Diese Funktion wartet auf den Empfang einer Message im MessagePort. In unserem Fall wird dies die Startup-Message der Workbench sein. Diese Message muß nun abgeholt werden, damit sie aus der Warteschlange der Messages entfernt wird. Dazu dient die GetMsg()-Funktion:

```
lea     $5c(a4),a0       ;RastPort-Adresse in A0
jsr     GetMsg(a6)       ;Message abholen
```

Sie können diese Message nun bei Bedarf auswerten. In D0 erhalten Sie aus der GetMsg()-Funktion einen Zeiger auf die Message-Struktur, die den Namen WBStartup trägt.

Diese Message enthält folgende Elemente:

Am Anfang liegt eine normale Message-Struktur. Darauf folgen die Elemente der eigentlichen Startup-Struktur:

Offset	Name	Bedeutung
\$14	sm_Process	Process-Descriptor
\$18	sm_Segment	Programmsegment-Descriptor
\$1C	sm_NumArgs	Anzahl der übergebenen Argumente
\$20	sm_ToolWindow	Beschreibung des zu öffnenden Fensters
\$24	sm_ArgList	Zeiger auf die Argumente selbst

sm_Arglist zeigt auf die Elemente der übergebenen Argumente. Diese Argumente beinhalten die Informationen über die zum Zeitpunkt des Programmstartes aktivierten Icons der Workbench. Einige Programme nutzen dies so, daß z.B. eine beim Aufruf des Programmes durch Shift-Klick zusätzlich aktivierte Text-Datei vom Programm geladen und ausgegeben wird. Die Argumente der Liste, auf die sm_ArgList zeigt, bestehen aus den Zeigern:

wa_Lock	Datei-Lock (Directory-Beschreibung)
wa_Name	Zeiger auf Dateinamen

Um die Anwendung und Programmierung dieser Message-Auswertung zu demonstrieren, wollen wir nun ein Programm schreiben, das die Tool-Types ermittelt und ausgibt. Diese Tool-Types sind die Eintragungen, die im Workbench-Programm INFO in die jeweilige Datei geschrieben werden können. Dafür selektieren Sie eine Datei (einmal Klick) und wählen dann aus dem 'Workbench'-Menü den Punkt Info an. Es öffnet sich dann ein Dialogfenster, in dem Sie in der Eingabemaske TOOL TYPES durch Anklicken von Add Eintragungen machen können. Diese Eintragungen werden von einigen Programmen für Voreinstellungen verwendet (z.B. Notepad).

Diese Daten werden in dem zum Programm gehörenden .info-File abgespeichert. In dieser Datei stehen außerdem noch die Daten für das Icon, seine Position im Fenster und vieles mehr. Um in einem Programm diese Daten zu bekommen, ist eine weitere Library auf der Workbench-Diskette im LIBS-Ordner enthalten: die Icon-Library.

Diese Library enthält nun Funktionen zur Bearbeitung der .info-Files. Eine davon ist die Funktion GetDiskObject(), welche die .info-Datei lädt und einen Zeiger auf deren Struktur zurückgibt. Diese Funktion verwendet auch unser Programm. Bevor wir auf die Einzelheiten der Icon-Library und der DiskObject-Struktur eingehen, möchte ich Ihnen dieses Programm vorstellen, da dies einiges anschaulicher machen kann:

```
;** Workbench-Message- und .info-Auswertungsdemo S.D. **
```

```
execbase = 4           ;EXEC-Basisadresse
FindTask = -294        ;Task suchen
WaitPort = -384        ;auf Message warten
GetMsg = -372          ;Message abholen

OpenLib = -408         ;Library öffnen
CloseLib = -414        ;Library schließen
Open = -30             ;Kanal öffnen
Close = -36            ;Kanal schließen
Read = -42             ;Daten einlesen
Write = -48            ;Daten ausgeben
CurrentDir = -126      ;aktuelles Directory setzen
mode_old = 1005        ;Modus für's Öffnen

GetDiskObject = -78    ;DiskObject laden
```

```
run:
```

```
    move.l execbase,a6      ;EXEC-Basisadresse

    suba.l a1,a1
    jsr    FindTask(a6)    ;eigenen Task suchen

    move.l d0,a4           ;Zeiger in A4
    tst.l  $ac(a4)         ;pr_CLI: CLI oder Workbench?
    bne    fromCLI        ;CLI! Ende...
```

```

        lea    $5c(a4),a0      ;WBench-Message
        jsr    WaitPort(a6)    ;abwarten

        jsr    GetMsg(a6)      ;Message abholen
        move.l d0,message      ;Zeiger retten

; **** Libraries und Fenster öffnen ****

        lea    iconname,a1     ;"icon.library"
        clr.l  d0
        jsr    OpenLib(a6)     ;ICON.library öffnen
        move.l d0,iconbase     ;Basis retten
        beq    ende3           ;Fehler aufgetreten!

        lea    dosname,a1      ;"dos.library"
        clr.l  d0
        jsr    OpenLib(a6)     ;DOS öffnen
        move.l d0,dosbase      ;Basis retten
        beq    ende2           ;Fehler aufgetreten!

        move.l d0,a6
        move.l #conname,d1
        move.l #mode_old,d2
        jsr    Open(a6)        ;CON-Fenster öffnen
        move.l d0,conbase
        beq    ende1           ;Fehler aufgetreten!

;**** Das aktuelle Directory setzen, wenn nötig ****

        move.l message,a0      ;Zeiger auf WbMessage
        move.l $24(a0),a0      ;sm_ArgList: Zeiger auf
                                ;Argumente
        beq    ende            ;keine Argumente!

        move.l (a0),d1         ;D1 => Lock
        move.l dosbase,a6
        jsr    CurrentDir(a6)  ;aktuelles Directory setzen

; **** Disk-Objekt (.info-Datei) laden ****

        move.l message,a0
        move.l $24(a0),a0      ;sm_ArgList-Zeiger
        move.l 4(a0),a0        ;wa_Name: Zeiger auf Namen

        move.l iconbase,a6
        jsr    GetDiskObject(a6) ;Disk-Objekt laden

```

; **** Tool-Type-Einträge im Fenster ausgeben ****

```
move.l d0,a1      ;Zeiger auf DiskObject-Struktur
move.l $36(a1),a1 ;do_ToolTypes: Zeiger auf
                  ;ToolType-Array
move.l a1,typetext ;Textzeiger retten
```

typesloop:

```
move.l typetext,a1 ;Textzeiger laden
move.l (a1)+,a0    ;Zeiger auf Text in A0
cmp.l  #0,a0       ;Text vorhanden?
beq    nomore      ;nein: Ende der Ausgaben
move.l a1,typetext ;sonst Zeiger retten
```

```
move.l a0,d2       ;= Textadresse für Ausgabe
move.l a0,d3       ;nun noch die Länge ermitteln
```

lenloop:

```
tst.b  (a0)+       ;Ende suchen
bne    lenloop
sub.l  a0,d3       ;Länge des Textes errechnen
not.l  d3          ;und korrigieren
```

```
move.l dosbase,a6
move.l conbase,d1
jsr    Write(a6)   ;Text im Fenster ausgeben
```

```
move.l conbase,d1
move.l #lf,d2      ;Linefeed:
move.l #1,d3
jsr    Write(a6)   ;nächste Zeile
```

```
bra    typesloop   ;auf zum nächsten Eintrag!
```

; **** Das waren alle, nun auf Taste warten ****

nomore:

```
move.l conbase,d1
move.l #1,d3       ;ein Zeichen
move.l #buffer,d2  ;in Buffer
jsr    Read(a6)    ;einlesen (auf Return warten)
```

; **** Programm-Ende: alles schließen und zurück ****

```
ende:  move.l conbase,d1
        move.l dosbase,a6
```

```

        jsr    Close(a6)          ;Fenster schließen
ende1:
        move.l execbase,a6
        move.l dosbase,a1
        jsr    Closeslib(a6)     ;DOS schließen
ende2:
        move.l iconbase,a1
        jsr    Closeslib(a6)     ;ICON.library schließen

fromCLI:
ende3:
        rts                      ;Programm-Ende

; **** Datenfelder ****

dosbase: blk.l 1      ;DOS-Basisadresse
conbase: blk.l 1      ;Fenster-Basis
iconbase: blk.l 1     ;icon.library-Basis
message: blk.l 1      ;Zeiger auf WbMessage
typetext: blk.l 1     ;Text-Zeiger

dosname: dc.b 'dos.library',0
iconname: dc.b 'icon.library',0
conname: dc.b 'CON:10/20/300/100/** Message-Ausgabe',0
lf:      dc.b $a
buffer:  blk.b 2

```

Dieses Programm funktioniert nur, wenn es von der Workbench aus gestartet wurde. Andernfalls wird einfach abgebrochen (fromCLI). Um es starten zu können, muß noch ein Icon für dieses Programm erstellt werden. Dies können Sie mit dem Icon-Editor auf einfache Weise erledigen. Danach muß es unter dem gleichen Namen wie obiges Programm abgespeichert werden, der Anhang .info wird automatisch erstellt.

Ist dies geschehen, so können Sie nun das Icon anklicken und im Workbench-Menü den Punkt Info wählen. In dem so erscheinenden Fenster können Sie nun einen oder mehrere Einträge in TOOL TYPES eingeben und mit SAVE abspeichern.

Wenn Sie dann Ihr Icon mit einem Doppelklick aktivieren, so wird das dazugehörige Programm geladen und gestartet. Das Programm führt dann die nötigen Schritte aus, um sowohl die

Workbench-Startup-Message (WBStartup) als auch die DiskObject-Struktur zu erhalten und auszuwerten.

Die Struktur des Disk-Objektes bzw. der .info-Datei ist folgendermaßen aufgebaut:

Offset	Name	Inhalt
0	do_Magic	Eine 'magische Zahl', die diese Datei als gültig erklärt (\$E310)
2	do_Version	Versionsnummer (1)
4	do_Gadget	Hier beginnt eine Gadget-Struktur, die das Aussehen und die Position des Icons bestimmt
\$30	do_Type	Objekt-Typ (Tool, Projekt etc.)
\$32	do_DefaultTool	Standard-Programm der Diskette
\$36	do_ToolTypes	Zeiger auf Text-Feld der Typen
\$3A	do_CurrentX	
\$3E	do_CurrentY	Icon-Position im Fenster
\$42	do_DrawerData	Zeiger auf die Unterdirectory-Fenster-Struktur
\$46	do_ToolWindow	Standard-Fenster für Tools
\$4A	do_StackSize	Stack-Größe für Tools

Der Zeiger do_ToolTypes zeigt auf eine Zeigerliste, deren Einträge wiederum auf die Texte der im Info-Fenster eingetragenen Tool-Types zeigen und mit einer Null enden. Diese Zeiger werden im Programm verwendet, um die Texte ausgeben zu können.

Aus dem Beispielprogramm können Sie also leicht entnehmen, wie Sie an die Tool-Types Ihres Programmes herankommen. Dort können grundsätzliche Eintragungen vorgenommen werden, die die Funktion des Programmes steuern sollen. Dies ist auch bei dem Notepad-Programm der Workbench realisiert, bei dem über die Tool-Types Parameter wie die Größe des Eingabefensters oder die zu verwendende Schrift bestimmt werden.

Die Eintragungen dort sind üblicherweise immer in der Form

NAME=<Parameter>[|<Parameter>]

eingetragen. Dies wird auch bei Notepad verlangt. Der Vorteil dieser Eintragsform ist einfach der, daß in der Icon-Library

zwei Funktionen vorgesehen sind, die diese Zeilen überprüfen können.

Die erste Funktion, `FindToolType()` mit dem Offset -96, sucht die Einträge der Tool-Types nach einem bestimmten Namen durch. Im Beispiel Notepad etwa wird nach einer Zeile mit dem Namen WINDOW gesucht. Es wird dann ein Zeiger auf die dem Gleichheitszeichen folgenden Parameter zurückgegeben oder eine Null, wenn keine Zeile mit diesem Namen vorkommt.

Dieser Zeiger wird dann der anderen Funktion, `MatchToolType()` mit dem Offset -102, zusammen mit einem weiteren Zeiger auf einen Vergleichsparameter übergeben. Der daraus resultierende Wert zeigt dann an, ob der Vergleichsparameter in der Zeile vorkommt oder nicht.

Dies wird z.B. dann benutzt, wenn ein Programm Dateien lesen kann, aber nur bestimmte Arten von Dateien lesen soll. Werden diese Arten in Tool-Types eingetragen, können sie nachher vom Programm mit dem zu ladenden Dateityp verglichen werden, um festzustellen, ob es diese Datei laden darf.

3.3.2 Aufbau der transienten CLI-Befehle

Wie Sie ja bereits wissen, sind die Befehle des normalen CLI alle transient, d.h. sie sind als Programme auf der Diskette im C-Unterdirectory abgespeichert. Wenn Sie also etwas im CLI eingeben, so wird geprüft, ob es sich um einen Dateinamen im aktuellen Directory oder ein Kommando handelt, dessen Name im C-Directory enthalten ist. Ist dies der Fall, so wird das entsprechende Programm aufgerufen.

Nahezu alle Kommandos benötigen dann für ihre Arbeit Zugang zur DOS-Bibliothek, um die gewünschte Funktion ausführen zu können. Damit aber nicht jedes dieser Programme extra die DOS-Bibliothek öffnen muß, werden den Programmen in den Prozessor-Registern einige Parameter übergeben.

Die Register D0 und A0 enthalten die Länge und Adresse des Parametertextes, der hinter dem Kommando eingegeben wurde. Dies wurde bereits am Beispiel des FONTS-Programmes erklärt.

Die anderen Register enthalten weitere interessante Werte:

Register	Inhalt
D0	Anzahl der Parameter-Zeichen
A0	Adresse des Parameter-Textes
A1	Zeiger auf Stack-Anfang
A2	Zeiger auf interne DOS-Bibliothek
A3	Zeiger auf Stackgröße
A4	Zeiger auf Programmbeginn
A5	Zeiger auf Routine zum Funktionsaufruf
A6	Zeiger auf Rücksprungroutine

Betrachten wir besonders die Register A2, A5 und A6. Mit diesen Registern kann man bereits ein CLI-Kommando schreiben, das ohne eigenes Öffnen der DOS-Bibliothek auskommt.

Die Konvention zum Aufruf dieser Routinen ist allerdings etwas anders als beim normalen DOS-Aufruf. Ab der Adresse, auf die A2 zeigt, liegen eine Reihe Sprungadressen, die auf die einzelnen DOS-Routinen zeigen. Aufgerufen werden sie allerdings nicht direkt, sondern mit der Adresse in A4 über JSR (A5). Der Rückgabeparameter wird nicht in D0, sondern in D1 übergeben. Auch die Offsets innerhalb der Tabelle sind andere als die des normalen Aufrufes.

Diese Offsets sind nicht absolut sicher, da sie nicht von Commodore dokumentiert sind. Dennoch sind die unten aufgeführten Offsets in der momentanen Amiga-Version korrekt.

Bevor wir die einzelnen Offsets der DOS-Funktionen im direkten Aufruf auflisten, sollte deren Anwendung erst einmal klar werden. Dafür folgt nun wieder ein kleines Programm, das nichts anderes tut als ein kleines Fenster zu öffnen, auf die Betätigung der Return-Taste zu warten und dann wieder das

Fenster zu schließen. Dies sind drei DOS-Funktionen, die ohne Öffnen der DOS-Bibliothek aufgerufen werden.

Für den Funktionsaufruf ist hier ein Macro verwendet worden. Dieses Macro wird beim Assemblieren des Programmes überall dort eingesetzt, wo der Macroname (doscall) auftaucht. Der danach angegebene Parameter wird dann dort eingesetzt, wo in der Macrodefinition ?1 steht. Dieser Parameter ist dann unser Offset.

```
;**** vom CLI: DOS-Grundfunktionen 6/87 S.D. ****
```

```
Open   =$ff      ;DOS-Kommando: Open
Close  =$5d      ;          Close
Read   =$fd      ;          Read
```

```
mode_old=1005
```

```
; **** Definition des Macros 'doscall' ****
```

```
doscall: MACRO          ; ** direkter DOS-Aufruf **
    move.b  #?1,d0
    ext     d0           ;Offset in Langwort
    ext.l   d0           ;umwandeln
    lsl     #2,d0
    move.l  0(a2,d0),a4 ;Funktions-Adresse
    moveq   #$c,d0
    jsr     (a5)         ;Funktions-Aufruf
ENDM
```

```
; **** Programmanfang ****
```

```
run:
    move.l  #consolename,d1 ;Consol-Definition
    move.l  #mode_old,d2    ;Modus
    doscall Open            ;CON:-Fenster öffnen
    move.l  d1,conhandle

    move.l  conhandle,d1
    move.l  #inbuff,d2      ;Buffer-Adresse
    move.l  #1,d3           ;1 Zeichen
    doscall Read            ;Zeichen einlesen (Return)

    move.l  conhandle,d1    ;Fenster schliessen
    doscall Close          ;mit Close
```



```

        clr.l    d0                ;Status: OK
        jsr      (a6)              ;Ende des Programmes

; **** Datenfelder ****

conhandle: dc.l 0
inbuff:    blk.b 8
consolname: dc.b 'RAW:100/50/300/100/** Test-Fenster',0
even

```

Sie sehen, wie einfach ein CLI-Kommando zu programmieren ist! Mit ganzen 12 Zeilen Programmtext werden schon 3 DOS-Funktionen ausgeführt. Auch das Fonts-Programm aus dem CLI-Kapitel wäre so wesentlich kürzer geworden. Probieren Sie es aus!

Die Funktions-Offsets der DOS-Funktionen sind, wie Sie sehen, andere als beim normalen DOS-Aufruf. Hier eine Liste der DOS-Kommandos mit den Offsets, die bei obiger Programmierung gültig sind:

Offset	Funktionsname	Offset	Funktionsname
\$FF	Open	\$EF	IoErr
\$5D	Close	\$EE	CreateProc
\$FD	Read	\$02	Exit
\$FA	Write	\$ED	LoadSeg
\$41	Input	\$52	UnLoadSeg
\$42	Output	\$EC	GetPacket
\$F8	Seek	\$EB	QueuePacket
\$F7	Delete	\$EA	DeviceProc
\$F6	Rename	\$E9	SetComment
\$F5	Lock	\$E8	SetProtect
\$6D	UnLock	\$E7	DateStamp
\$71	DupLock	\$2F	Delay
\$F4	Examine	\$57	WaitForChar
\$F3	ExNext	\$23	ParentDir
\$F2	Info	\$E6	IsInteractive
\$F1	CreateDir	\$E5	Execute
\$F0	CurrentDir		

Dies sind, wie Sie an dem Programm-Beispiel sehen können, eigentlich keine Offsets, sondern die Nummern der zu verwen-

denden Vektoren der durch A2 angezeigten Tabelle. Dabei sind die Werte über \$7F negative Werte, also wird eine Adresse unterhalb der Adresse in A2 verwendet.

3.4 Ein-/Ausgabe

Ein sehr wichtiger Bestandteil eines Programmes ist der Datenaustausch mit der Umwelt, z.B. über Bildschirm, Tastatur, Disketten oder der Schnittstellen. Diese Ein-/Ausgabe, auch einfach I/O (Input/Output) genannt, befähigt ein Programm erst zu der vollen Ausnutzung des Computers, auf dem es läuft. Um dies zu bewerkstelligen, gibt es grundsätzlich drei Möglichkeiten.

Die erste wäre die Ein-/Ausgabe mittels der entsprechenden DOS-Funktionen wie Open(), Close(), Read() und Write(). Diese Methode ist die deutlich einfachere, da Sie bei der Programmierung keinen großen Aufwand treiben müssen. Der Nachteil ist allerdings, daß die aufgerufene Funktion erst vollständig erledigt wird, bevor Ihr Programm weiterarbeiten kann.

Diesen Nachteil hat die zweite Methode nicht. Das Zauberwort heißt hier Devices. Mit diesen Devices können Sie die gesamte Ein-/Ausgabe vollständig selbständig ablaufen lassen, während Ihr Programm weiterarbeitet. Sie läuft dabei im Hintergrund, also parallel zu Ihrem Programm, und kostet daher kaum wertvolle Rechenzeit. Der Nachteil dieser Technik ist allerdings der wesentlich höhere Programmieraufwand, der dafür benötigt wird.

Die dritte Methode der Ein-/Ausgabe ist schließlich die direkte Programmierung der Hardware des Amiga. Dies setzt jedoch sehr genaue Kenntnisse des Systems voraus und hat zudem den Nachteil, daß diese Methode im Multitasking-Betrieb zu großen Komplikationen führen kann. Im Hardwareteil dieses Buches finden Sie dazu mehr Informationen.

Beginnen wir die Betrachtung der Ein-/Ausgabe-Programmierung bei der Standard-Methode: der Verwendung der DOS-Funktionen.

3.4.1 Standard-I/O

Wie bereits erwähnt, sind die vier DOS-Funktionen `Open()`, `Close()`, `Read()` und `Write()` für die Ein-/Ausgabe von Daten zuständig. Mit ihnen können die meisten Funktionen ausgeführt werden, die ein Programm benötigt.

Es stehen eine Reihe von Ein-/Ausgabekanälen zur Verfügung, die das DOS bereits mit Namen kennt. Diese Namen können dann in einem `Open`-Befehl verwendet werden. Die Standard-Kanäle heißen:

- DFn:** Bezeichnet das Diskettenlaufwerk mit der Nummer *n*. Die Nummer *n* kann dabei 0, 1, 2 oder 3 sein.
- SYS:** Bezeichnet das Diskettenlaufwerk, von dem das System geladen wurde.
- RAM:** Steht für die RAM-Disk, die stets verfügbar ist und ihre Größe den enthaltenen Daten anpaßt. Sie kann wie ein Diskettenlaufwerk verwendet werden, nur daß die Daten nicht auf Diskette, sondern im RAM-Speicher des Amiga abgelegt werden.
- NIL:** Dieser Kanal ist ein wahres Datengrab: die hierin geschriebenen Daten werden verworfen und bewirken nichts. Dies ist manchmal recht brauchbar, wenn z.B. ein Programm Ausgaben machen will, die Sie jedoch nicht haben wollen.
- SER:** Steht für die serielle Schnittstelle (RS232) und bietet die Ein- und Ausgabe von Daten über diesen Port.
- PAR:** Bezeichnet die parallele Schnittstelle (Drucker-Port), welche ja 8 Ein-/Ausgabeleitungen enthält. Sie können hier also parallel anliegende Daten direkt einlesen oder ausgeben.

PRT: Steht ebenfalls für die parallele Schnittstelle, nur daß mit diesem Kanal ein Drucker angesprochen werden kann. Ist der Drucker jedoch für die serielle Schnittstelle definiert, so wird diese hierdurch angesprochen. Die Definitionen für den Drucker können Sie mit dem Preferences-Programm vorgeben.

CON: Gibt ein Fenster für die Ein-Ausgaben vor. Dieses Fenster wird beim Öffnen des Kanals automatisch geöffnet. Die Fensterparameter werden folgendermaßen angegeben:

CON:x/y/b/h/Name

x und y stellen die Koordinaten der linken oberen Ecke des Fensters auf dem Screen dar, b und h die Breite und Höhe des Fensters in Bildschirmpunkten und Name den Fenstertitel, der in der Titelzeile erscheinen wird. So definiert

CON:20/10/200/100/Test-Fenster

ein Fenster mit Namen 'Test-Fenster', welches an der Position x=20 und y=10 beginnt, 200 Punkte breit und 100 Punkte hoch ist.

RAW: Stellt ebenfalls ein Fenster dar und gibt die Ein- und Ausgaben an dieses Fenster weiter. Im Gegensatz zu CON: werden hier jedoch keine Funktionen vorbearbeitet (z.B. Editieren einer Zeile), so daß mit diesem Fenster nur auf ganz besondere Art und Weise gearbeitet werden kann.

***** Steht schließlich für das aktuelle Fenster, was z.B. das CLI-Fenster darstellt.

Beginnen wir mit der wohl wichtigsten Anwendung: der Tastatur-Eingabe und Bildschirm-Ausgabe.

3.4.1.1 Tastatur und Bildschirm

Wie Sie aus obiger Tabelle entnehmen können, stellt das AmigaDOS drei Möglichkeiten für die Bildschirm-Ein-/Ausgabe zur Verfügung: CON:, RAW: und *.

CON:-Fenster

Um ein CON:-Fenster zu öffnen, wird die Open()-Funktion des DOS verwendet. Die Funktion erwartet als Parameter einen Zeiger auf den Namen des zu öffnenden Kanals und den Modus, unter dem dieser geöffnet werden soll. Als Modus kommen in Frage:

Mode_new	für einen Kanal, in den nur geschrieben werden kann,
Mode_old	für einen Kanal, aus dem auch gelesen werden kann, und
Mode_readwrite	in der DOS-Version 1.2, bei der der Kanal sowohl beschrieben als auch ausgelesen werden kann.

Für das Öffnen eines CON:- oder RAW:-Fensters wird der Modus Mode_old verwendet, da der Kanal ja eigentlich bekannt ist und aus ihm auch gelesen werden kann.

Um dies zu demonstrieren, hier ein kleines Maschinenprogramm, das ein CON:-Fenster öffnet, einen Text darin ausgibt, auf eine Eingabe wartet und dann das Fenster wieder schließt:

```
;***** einfache CON:-Ein-/Ausgabe *****
```

```
OpenLib = -408  
closelib = -414  
ExecBase = 4
```

```
; Amiga DOS Offsets
```

```
Open      = -30  
Close     = -36
```

Read = -42
 Write = -48
 Exit = -144

Mode_old = 1005

```
run:
    move.l    excbase,a6        ;Zeiger auf EXEC-Bibliothek
    lea      dosname,a1
    moveq     #0,d0
    jsr      openlib(a6)        ;Open DOS-Library
    move.l    d0,dosbase
    beq       error            ;nicht geklappt

    move.l    dosbase,a6        ;DOS-Basisadresse in A6

    move.l    #name,d1          ;Zeiger auf Namen
    move.l    #mode_old,d2      ;Modus
    jsr      Open(a6)           ;Fenster öffnen
    move.l    d0,conhandle      ;Handle retten
    beq       error

    move.l    conhandle,d1      ;Fenster-Handle in D1
    move.l    #text,d2          ;Text-Adresse in D2
    move.l    #tende-text,d3    ;Länge in D3
    jsr      Write(a6)          ;Text ausgeben

    move.l    conhandle,d1      ;Fenster-Handle
    move.l    #buffer,d2        ;Puffer-Adresse
    move.l    #80,d3            ;max. Länge
    jsr      Read(a6)           ;Eingabe erwarten

    move.l    conhandle,d1
    jsr      Close(a6)          ;Fenster schließen

    bra ende                    ;fertig

error:
    move.l    #-1,d0            ;Error-Status
ende:
    move.l    d0,d1
    move.l    dosbase,a6
    jsr      Exit(a6)           ;Ende des Programmes

    rts                        ;kommt nicht vor
```

```
dosname:    dc.b 'dos.library',0
name:       dc.b 'CON:20/10/200/100/** Test-Fenster',0
text:       dc.b 'Bitte etwas eingeben !',0
tende:
buffer:     blk.b 80
even
dosbase:    dc.l 0
conhandle:  dc.l 0
```

RAW:-Fenster

Das oben aufgeführte Programm kann auch mit RAW: anstelle von CON: gestartet werden. Wenn Sie dies ausprobieren, so werden Sie den Unterschied sofort bemerken. Während die CON:-Version nämlich auf die Betätigung der Return-Taste nach der Eingabe wartet, kehrt das Programm mit dem RAW:-Fenster sofort nach der Betätigung irgendeiner Taste zurück. Dies gilt auch für die Funktions- oder Cursortasten, die ja vom CON:-Fenster nicht erkannt werden.

Ein CON:-Fenster bietet somit wesentlich mehr Komfort bei der Eingabe ganzer Texte, ein RAW:-Fenster dagegen macht die gesamte Tastatur verfügbar.

Aber nicht nur die normale Zeichendarstellung ist bei beiden Fensterarten möglich. Es gibt auch noch die Möglichkeit, andere Schriftarten darzustellen, wie z.B. unterstrichen oder fettgedruckt. Des weiteren lassen sich noch andere Funktionen auslösen, mit denen man das Fenster manipulieren kann. So läßt sich das Bild löschen, hoch- oder runterschieben usw. All diese Funktionen werden durch Steuersequenzen ausgelöst, die teilweise mit Parametern versehen in dem Fenster ausgegeben werden müssen.

Hier nun eine Liste der Steuerzeichen, welche eine Funktion auslösen. Diese Zeichen sind als Sedezimalzahlen aufgeführt.

Sequenz	Funktion
08	Backspace
0A	Linefeed, Cursor runter
0B	Cursor eine Zeile hoch
0C	Fenster löschen
0D	Carriage Return, Cursor in die erste Spalte
0E	auf Normaldarstellung schalten (0F zurücknehmen)
0F	auf Sonderzeichen schalten
1B	Escape

Die folgenden Sequenzen beginnen mit dem Zeichen \$9B, dem CSI (Control Sequence Introductor). Die auf dieses Zeichen folgenden Zeichen bewirken dann eine Funktion. Die in eckigen Klammern angegebenen Werte können entfallen. Die Angabe für 'n' ist in ASCII-Zeichen als ein- oder mehrstellige dezimale Zahl anzugeben. Der Wert, der bei weggelassener Angabe für n angenommen wird, ist hier in Klammern hinter n angegeben.

Sequenz	Funktion
9B [n] 40	n Leerzeichen einschieben
9B [n] 41	Cursor um n (1) Zeilen hoch
9B [n] 42	Cursor um n (1) Zeilen runter
9B [n] 43	Cursor um n (1) Zeilen rechts
9B [n] 44	Cursor um n (1) Zeilen links
9B [n] 45	Cursor um n (1) Zeilen runter in Spalte 1
9B [n] 46	Cursor um n (1) Zeilen hoch in Spalte 1
9B [n] [3B n] 48	Cursor in Zeile; Spalte setzen
9B 4A	Fenster ab Cursor löschen
9B 4B	Zeile ab Cursor löschen
9B 4C	Zeile einfügen
9B 4D	Zeile löschen
9B [n] 50	n Zeichen ab Cursor löschen
9B [n] 53	n Zeilen hochschieben
9B [n] 54	n Zeilen runterschieben
9B 32 30 68	ab sofort: Linefeed => Linefeed+Return
9B 32 30 6C	ab sofort: Linefeed => nur Linefeed
9B 6E	sende die Cursor-Position! Es wird eine Zeichenkette folgender Form zurückgegeben:

9B (Zeile) 3B (Spalte) 52

9B (Stil);(Vordergrundfarbe);(Hintergrundfarbe) 6D

Die drei Parameter sind ebenfalls Dezimalzahlen im ASCII-Format. Sie bedeuten:

Stil: 0=normal
1=fett
3=italic (schräg)
4=unterstrichen
7=invers-Schrift

Vordergrundfarbe:

30-37: Farben 0-7 für Text

Hintergrundfarbe:

40-47: Farben 0-7 für Hintergrund

9B (Länge) 74 setzt die maximale Anzahl der darzustellenden Zeilen fest
9B (Breite) 75 setzt die maximale Zeilenlänge fest
9B (Abstand) 78 definiert den Abstand in Pixeln vom linken Rand des Fensters, ab dem die Ausgabe beginnen soll
9B (Abstand) 79 definiert den Abstand in Pixeln vom oberen Rand des Fensters, ab dem ausgegeben werden soll

Die letzten 4 Funktionen können durch Weglassen des Parameters auf die Normalstellung gebracht werden.

9B 30 20 70 Cursor unsichtbar machen
9B 20 70 Cursor sichtbar machen
9B 71 sende Fenster-Ausmaß! Es wird eine Zeichenkette folgender Form zurückgegeben:

9B 31 3B 31 3B (Zeilen) 3B (Spalten) 73

Um die Anwendung dieser Steuerzeichen zu demonstrieren, lassen Sie doch einmal folgenden Text in Ihrem Fenster ausgeben:

```
text: dc.b $9b,"4;31;40m"
      dc.b "Überschrift"
      dc.b $9b,"3;33;40m",$9b,"5;20H"
      dc.b "*** Hallo, Welt ! ***",0
```

Die Parameter für die Steuersequenzen sind hier, durch die Anführungszeichen markiert, einfach als ASCII-Zeichenketten

angegeben. Wie Sie sehen, können Sie so recht wirkungsvolle Textausgaben realisieren!

Ebenso wie diese Sequenzen gesendet werden können, werden diese auch empfangen, wenn eine Funktionstaste der Tastatur bzw. eine Cursortaste gedrückt wurde. Die Zeichen, die man in diesem Fall empfängt, sind folgende (<CSI> steht für \$9B):

Taste	ohne Shift	mit Shift
F1	<CSI>0~	<CSI>10~
F2	<CSI>1~	<CSI>11~
...		
F9	<CSI>8~	<CSI>18~
F10	<CSI>9~	<CSI>19~
HELP	<CSI>?~	<CSI>?~
hoch	<CSI>A	<CSI>T~
runter	<CSI>B	<CSI>S~
links	<CSI>C	<CSI> A~
rechts	<CSI>D	<CSI> @~

Auf diese Weise kann man also fast alles erfahren, was der Benutzer mit der Tastatur anfängt. Wenn das immer noch nicht ausreicht, so gibt es eine weitere Informationsquelle: die RAW-Input-Events. Dies sind Ereignisse, die auf Wunsch durch eine Sequenz gemeldet werden. Der Wunsch nach der Meldung dieser Ereignisse wird dem DOS wiederum durch eine Sequenz mitgeteilt, die so aussieht:

<CSI>n{

Das 'n' steht dabei für eine Zahl zwischen 1 und 16, die dem Ereignis entspricht, das gemeldet werden soll. Diese Ereignisse sind folgende:

- 1 Taste gedrückt
- 2 Maustaste gedrückt
- 3 Fenster wurde aktiviert
- 4 Maus verschoben
- 5 unbenutzt
- 6 Timer
- 7 Gadget angewählt
- 8 Gadget losgelassen
- 9 Requester eingeblendet
- 10 Menü angewählt
- 11 Fenster geschlossen (s. Console-Device)
- 12 Fenstergröße verändert
- 13 Fenster erneuert
- 14 Einstellungen verändert
- 15 Diskette aus dem Laufwerk genommen
- 16 Diskette eingelegt

Einige dieser Ereignisse (10, 11) sind in unserem Fall nicht verfügbar, da ein mit DOS geöffnetes Fenster ja weder über Menüs noch über das Schließsymbol verfügt. Diese Dinge werden aber dann wieder interessant, wenn man sein Consolen-Fenster selbst gestaltet hat. Dies wird allerdings nur über die Kombination Intuition und Devices möglich und wird daher später im Consol-Device-Kapitel extra behandelt.

Wenn nun ein solchermaßen gewünschtes Ereignis eintritt (Diskette eingelegt o.ä.), so wird eine Sequenz folgenden Formats gesendet:

```
<CSI><Klasse>;<Unterklasse>;<Taste>;<Status>;<X>;<Y>;  
    <Sekunden>;<Mikrosekunden>|
```

Dabei bedeutet:

CSI den Control-Sequence-Introducer \$9B
 Klasse die Ereignis-Nummer (s.o.)
 Unterklasse wird nicht verwendet und ist immer 0
 Taste Tastaturcode der letzten Taste oder Maustaste
 Status Tastaturstatus:

Bit	Maske	Bedeutung
0	0001	linke Shift-Taste
1	0002	rechte Shift-Taste
2	0004	CapsLock-Taste
3	0008	Control
4	0010	linke Alternate-Taste
5	0020	rechte Alternate-Taste
6	0040	linke Amiga-Taste
7	0080	rechte Amiga-Taste
8	0100	Ziffernblock
9	0200	Tastenwiederholung
10	0400	Interrupt (unbenutzt)
11	0800	aktives Fenster
12	1000	linker Mausknopf
13	2000	rechter Mausknopf
14	4000	mittlerer Mausknopf (unbenutzt)
15	8000	relative Mauskoordinaten

X und Y Koordinaten des Mauszeigers bei einem Maus-Ereignis

Sekunden

Mikrosekunden Systemzeit bei dem Ereignis

Die Werte, die auf diese Weise erhalten werden, sind Dezimalzahlen im ASCII-Code. Wenn Sie diese Werte also im Programm auswerten wollen, so müssen Sie diese erst umwandeln.

***-Fenster**

Die meisten CLI-Kommandos verwenden *, da dies die einfachste Art ist. Da dieses das aktuelle Fenster angibt, das natürlich schon geöffnet ist, kann dabei sogar das Öffnen und Schließen des Kanals entfallen.

Da die Funktionen Read() und Write() dennoch ein Handle des Kanals benötigen, in den oder aus dem sie Daten lesen sollen, muß dies zuerst einmal ermittelt werden.

Für diesen Zweck sind die DOS-Funktionen Input() und Output() vorgesehen. Diese Funktionen benötigen keine Parameter und geben das Handle des entsprechenden Standard-Kanals zurück. Dies wird das CLI-Fenster sein, wenn das Programm einfach von dort aus aufgerufen wurde. Wurde beim Aufruf jedoch die Eingabe oder Ausgabe mit der <- bzw. >-Funktion des CLI umgeleitet, so wird das dadurch aktuelle Handle von der Input()- bzw Output()-Funktion ermittelt.

3.4.1.2 Disketten-Dateien

Auf die gleiche Art wie CON:- oder RAW:-Fenster können auch Disketten-Dateien geöffnet und bearbeitet werden, wobei nur einige Dinge dazukommen. So spielt der beim Öffnen übergebene Modus eine große Rolle: wird 'Modus alt' übergeben, so wird eine bestehende Datei auf der Diskette gesucht, aus welcher auch nur gelesen werden kann. Bei 'Modus neu' wird die Datei neu angelegt, eine bereits unter diesem Namen existierende Datei wird gelöscht. In die so geöffnete Datei kann nur geschrieben werden. Beim 'Modus readwrite' kann eine bereits bestehende Datei sowohl gelesen als auch beschrieben, also verändert werden.

Für die DOS-Funktionen Read(), Write() und Close() ändert sich gegenüber der Bildschirm-Ein-/Ausgabe nichts. Es kommen aber einige Funktionen dazu, die bei der Bedienung von Diskettendateien sehr nützlich sind.

Da man aus einer Datei Daten nach und nach auslesen kann, muß sich das System merken, an welcher Stelle innerhalb der Datei zuletzt zugegriffen wurde. Dies wird durch einen Zeiger bewirkt, welchen man auch direkt verstellen kann. Dazu dient die `Seek()`-Funktion, mit der man den Zeiger beliebig innerhalb der Datei vorwärts und rückwärts verschieben kann. Dabei wird eine absolute Position angegeben, die entweder relativ zur aktuellen Position, zum Dateianfang oder zum Dateiende zählt.

Eine weitere Funktion des DOS erlaubt es, eine Datei von der Diskette zu löschen: die `Delete()`-Funktion. Mit ihr können auch Unterdirectories gelöscht werden, allerdings nur, wenn diese leer sind.

Die Namen der Dateien sind ebenfalls veränderbar, und zwar mit der `Rename()`-Funktion. Hier wird einfach der alte und der neue Dateiname übergeben. Interessant bei dieser Funktion ist es, daß man nicht nur einfach den Namen einer Funktion, sondern auch ihre Position innerhalb der Diskette verändern kann. Wird nämlich im neuen Namen ein anderer Suchpfad angegeben, so wird die Datei in dieses andere Unterdirectory 'verschoben' (nicht kopiert). Dies funktioniert allerdings nur auf einer Diskette, die Angabe eines anderen Diskettennamens in neuen Namen führt zu einer Fehlermeldung.

Eine Diskettendatei kann zusätzlich gegen verschiedene Funktionen geschützt werden. Dies wird durch die Maske bestimmt, die der `SetProtection()`-Funktion übergeben wird. Die ersten 4 Bits dieser Maske (Bit 0-3) geben jeweils an, ob die Datei gegen folgende Aktionen geschützt ist:

Bit	Bedeutung, wenn gesetzt
-----	-------------------------

0	Datei nicht löschrbar
1	Datei nicht ausführbar
2	Datei nicht überschreibbar
3	Datei nicht lesbar

3.4.1.3 Serielle Schnittstelle

Die serielle Schnittstelle kann ebenso wie etwa Bildschirm-Ein-/Ausgaben behandelt werden. Es wird ein Kanal mit dem Namen SER: geöffnet und in diesen Kanal geschrieben bzw. aus ihm gelesen. Dabei können jedoch drei Probleme auftauchen:

- 1) Beim Aufruf der Read()-Funktion wartet der Amiga auf den Empfang von einem oder mehreren Zeichen von der seriellen Schnittstelle. Kommen jedoch dort keine an, so wartet der Amiga, vergebens. Aus diesem Grund sollte man in einem Programm, das Daten von dieser Schnittstelle holen will und nicht absolut sicher ist, daß dort auch welche kommen, besser vor der Read()- die WaitForChar()-Funktion aufrufen. Mit dieser Funktion kann eine beliebige Zeit (anzugeben in Mikrosekunden) auf den Empfang gewartet werden. Kommt in dieser Zeit nichts an, so wird eine Null zurückgegeben, und das Programm kann ggf. eine Fehlermeldung ausgeben und aufgeben. Wenn doch etwas angekommen ist, so erhält man den Wert -1 und kann nun mit dem Lesen beginnen.
- 2) Es werden Daten empfangen, ohne daß bekannt ist, wie viele Daten kommen werden. Dabei kann das unter 1) beschriebene Problem auftreten. Hier liegt auch der Grund dafür, weshalb man nicht vom CLI aus mit z.B. COPY SER: TO * von der seriellen Schnittstelle ankommende Daten ansehen kann. Das CLI weiß ja nicht, wann die Daten anfangen oder aufhören, und streikt. Einen solchen Befehl kann man dann leider nur noch mit Reset beenden.
- 3) Ein Programm will über die Schnittstelle Daten senden oder empfangen, deren Einstellungen jedoch nicht stimmen. Man kann zwar dann mit dem Preferences-Programm diese Einstellungen vornehmen und neu starten, was jedoch sehr lästig ist. Ebenso wie das Preferences-Programm kann natürlich auch Ihr Programm diese Einstellungen selbst vornehmen. Dies ist jedoch nicht mit einem einfachen DOS-Kommando möglich, sondern muß über die I/O-Funktionen mit dem Serial-Device erledigt

werden, deren Behandlung Sie im entsprechenden Kapitel finden.

3.4.1.4 Parallele Schnittstelle

Die Programmierung der parallelen Schnittstelle (PAR:) ist normalerweise nicht notwendig, da dort meist der Drucker angeschlossen ist. Dennoch ist diese Schnittstelle recht interessant, da man dort nicht nur Daten ausgeben, sondern auch einlesen kann.

Die wohl einfachste Art der Programmierung gerade dieser Schnittstelle ist der direkte Weg über die Hardware-Register. Dies hat allerdings den Nachteil, daß eventuell Probleme beim Multitasking auftreten können, wenn ein anderes Programm auch auf diese Schnittstelle zugreifen will. Aus diesem Grund ist es sicherer, über das DOS zuzugreifen. Dabei ist allerdings auch das Datenformat vorgeschrieben, und die Möglichkeit, einzelne Bits als Eingang und andere als Ausgang zu programmieren, entfällt.

4. Devices

In den Devices liegt eine der großen Stärken des Amiga-Betriebssystems. Es handelt sich dabei um Programmpakete, die einige Aufgaben übernehmen. Diese Aufgaben werden den Devices von einem laufenden Programm erteilt, das dann entweder auf ein Ergebnis wartet oder weiterarbeiten kann. Auf diese Weise kann ein Programm relativ einfach das Multitasking verwenden.

Der grundsätzliche Aufbau solcher Devices wurde bereits im EXEC-Kapitel erklärt. Wir wollen uns nun mit der praktischen Anwendung befassen. Zunächst ein Blick auf die Vorgehensweise bei der Programmierung, die in folgenden Schritten abläuft:

1. Da die Erledigung einer Aufgabe von der Device mittels einer Message gemeldet wird, muß der Empfänger dieser Message, eben das eigene Programm, zunächst einmal ermittelt werden. Dies wird mit der FindTask()-Funktion des EXEC erledigt, der man als Parameter eine Null übergibt. Der so erhaltene Wert wird dann im nächsten Schritt verwendet.
2. Für die Message der Device wird ein Port mittels der AddPort()-Funktion angelegt. Es handelt sich dabei um einen Reply-Port, was soviel wie Antwort-Port heißt. In dieser MessagePort-Struktur wird im Eintrag SigTask (Portadresse +\$10) der eben erhaltene Zeiger auf die eigene Task-Struktur abgelegt.
3. Die Device wird mittels der OpenDevice()-Funktion geöffnet. Dabei muß ein Zeiger auf den Device-Namen und einer auf die I/O-Struktur übergeben werden, die dann zur Kommunikation mit dem Device-Programm benötigt wird.

4. In der I/O-Struktur werden die Parameter eingetragen, die für die gewünschte Funktion nötig sind. Die Anzahl und Art der einzutragenden Parameter sind je nach Funktion sehr unterschiedlich.
5. Die Arbeit der Device wird mit DoIo() oder SendIo() gestartet. Bei DoIo() wartet das aufrufende Programm auf das OK-Signal im Reply-Port, mit SendIo() wird die Arbeit der Device nur gestartet, und das Programm kann weiterarbeiten.

Hier sind zwei Strukturen aufgetaucht, die die Kommunikation zwischen Anwender-Programm und Devices steuern. Dies sind die Port- und die I/O-Struktur, die an anderer Stelle bereits beschrieben wurden. Hier noch einmal die Standard-Struktur der I/O-Operationen:

Offset	Name	Bedeutung
STRUCT MsgNode		
0	Succ	Zeiger auf nachfolgenden Eintrag
4	Pred	Zeiger auf vorhergehender Eintrag
8	Type	Eintragstyp
9	Pri	Priorität
10	Name	Zeiger auf Namen
14	ReplyPort	Zeiger auf ReplyPort
18	MNLength	Node-Länge
STRUCT IOExt		
20	IO_DEVICE	Zeiger auf Device-Node
24	IO_UNIT	interne Einheitsnummer
28	IO_COMMAND	Kommando
30	IO_FLAGS	Flags
31	IO_ERROR	Fehlerstatus
STRUCT IOStdExt		
32	IO_ACTUAL	Anzahl der übertragenen Bytes
36	IO_LENGTH	Anzahl der zu übertragenden Bytes
40	IO_DATA	Zeiger auf Datenpuffer
44	IO_OFFSET	Offset (z.B. für Trackdisk-Device)
48	Hier beginnt die jeweilige erweiterte Struktur	

Die normalen I/O-Funktionen werden mit den Standard-Kommandos ausgelöst, die zu den I/O-Definitionen gehören. Diese Kommandos sind:

CMD_INVALID	(0)	ungültiges Kommando
CMD_RESET	(1)	Zurücksetzen der Device auf den Anfangszustand
CMD_READ	(2)	Lesen aus der Device
CMD_WRITE	(3)	Schreiben in die Device
CMD_UPDATE	(4)	Aufarbeiten der Puffer
CMD_CLEAR	(5)	Löschen aller Puffer
CMD_STOP	(6)	Pause einlegen
CMD_START	(7)	nach der Pause weiterarbeiten
CMD_FLUSH	(8)	Abbruch der momentanen Arbeit

Zusätzlich zu diesen Kommandos gibt es für jede Device einige weitere, die in den folgenden Beispielen erklärt werden.

Auf einer normalen Workbench-Diskette befinden sich etliche Devices im DEVS-Unterdirectory. Einige weitere Devices sind nicht auf der Diskette zu finden und trotzdem verfügbar, da sie resident im Amiga liegen.

Wir wollen uns nun mit der Programmierung der wichtigsten Devices anhand von Beispielen beschäftigen, die Sie in eigenen Programmen vielleicht gut gebrauchen können.

4.1 Trackdisk-Device: Zugriff auf Disketten

Das Trackdisk-Device ist die vom Betriebssystem vorgesehene Verbindung zu den Disketten. Dies wird auch vom DOS verwendet. Es bietet die Möglichkeit des direkten Disketten-Zugriffes, ohne auf die Hardware-Register zugreifen zu müssen.

Die erweiterte I/O-Struktur enthält folgende zwei Einträge (Langworte), welche allerdings nur für die erweiterten Kommandos nötig sind:

IOTD_COUNT	Anzahl der erlaubten Diskettenwechsel
IOTD_SECLABEL	Zeiger auf Sektor-Header-Feld, das pro zu lesenden Sektor 16 Bytes groß sein muß

Diese Device besitzt eine große Anzahl von zusätzlichen Kommandos. Dabei wird noch zwischen den normalen und den erweiterten Trackdisk-Kommandos unterschieden. Hier eine Liste aller gültigen Trackdisk-Kommandos:

Standard-Kommandos:

CMD_READ	(2)	Lesen eines oder mehrerer Sektoren von Diskette
CMD_WRITE	(3)	Schreiben eines oder mehrerer Sektoren auf Diskette
CMD_UPDATE	(4)	Trackpuffer auf Diskette zurückschreiben
CMD_CLEAR	(5)	Trackpuffer als ungültig erklären

Trackdisk-Kommandos:

TD_MOTOR	(9)	Ein-/Ausschalten eines Laufwerksmotors
TD_SEEK	(10)	Schreib-/Lesekopf des Laufwerks auf einen bestimmten Track positionieren
TD_FORMAT	(11)	Initialisierung eines oder mehrerer Tracks
TD_REMOVE	(12)	Interrupt-Routine installieren, die bei einem Diskettenwechsel aufgerufen wird.
TD_CHANGENUM	(13)	Anzahl der Diskettenwechsel ermitteln

TD_CHANGESTATE	(14)	Feststellen, ob eine Diskette eingelegt ist
TD_PROTSTATUS	(15)	Feststellen, ob die Diskette schreibgeschützt ist
TD_RAWREAD	(16)	Lesen des unbearbeiteten Disketteninhalts
TD_RAWWRITE	(17)	unbearbeitete Daten auf Diskette schreiben
TD_GETDRIVETYPE	(18)	Laufwerkstyp ermitteln (1=3½, 2=5¼ Zoll)
TD_GETNUMTRACKS	(19)	Gesamtzahl der Tracks ermitteln
TD_ADDCHANGEINT	(20)	Interrupt-Routine installieren, die bei Diskettenwechsel aufgerufen wird
TD_REMCHANGEINT	(21)	obige Routine abschalten
TD_LASTCOMM	(22)	letztes Kommando ermitteln

Erweiterte Kommandos (Nummern alle +32768 bzw. +\$8000):

Gleiche Funktion wie oben, jedoch nur, wenn kein Disketten-Wechsel vorgenommen wurde.

ETD_READ	(2)
ETD_WRITE	(3)
ETD_UPDATE	(4)
ETD_CLEAR	(5)
ETD_MOTOR	(9)
ETD_SEEK	(10)
ETD_FORMAT	(11)
ETD_RAWREAD	(16)
ETD_RAWWRITE	(17)

Wir wollen uns nun ein kleines Maschinenprogramm ansehen, das die Trackdisk-Device anwendet. Ein einfaches Beispiel ist es hierfür, ein paar Sektoren von der Diskette in den Speicher zu laden. Wenn Sie im Besitz eines Assembler-/Debugger-Paketes wie z.B. dem Profimat oder dem K-SEKA sind, so können Sie sich das Ergebnis direkt ansehen. Andernfalls können Sie die erhaltenen Daten ja auch mittels des Open()- und Write()-

Kommandos des AmigaDOS als Datei auf die Diskette schreiben und nachher mit TYPE und der Option H auf dem Bildschirm bzw. Drucker ausgeben lassen.

;*** Trackdisk-Device-Demo: Sektoren lesen 6/87 S.D. ***

```

ExecBase = 4           ;EXEC-Basisadresse
FindTask = -294        ;Task-Struktur suchen
AddPort   = -354        ;Port erstellen
RemPort   = -360        ;Port entfernen
OpenLib   = -408        ;Library öffnen
CloseLib  = -414        ;Library schließen
OpenDev   = -444        ;Device öffnen
CloseDev  = -450        ;Device schließen
DoIo      = -456        ;I/O starten und warten

run:
    move.l  execbase,a6    ;Zeiger auf EXEC-Bibliothek
    sub.l   a1,a1          ;eigener Task
    jsr     FindTask(a6)   ;Task suchen
    move.l  d0,readreply+$10 ;SigTask: eigener Task

    lea     readreply,a1
    jsr     AddPort(a6)    ;Add Reply-Port

    lea     diskio,a1      ;I/O-Struktur
    move.l  #0,d0          ;Laufwerk DF0:
    clr.l   d1             ;keine Flags
    lea     trddevice,a0   ;Device-Name
    jsr     OpenDev(a6)    ;Open trackdisk.device
    tst.l   d0             ;OK?
    bne     error          ;nein: Fehler aufgetreten!

    lea     diskio,a1
    move.l  #readreply,14(a1) ;set Reply-Port
    move    #2,28(a1)       ;Command: READ
    move.l  #diskbuff,40(a1) ;Puffer
    move.l  #2*512,36(a1)   ;Länge: 2 Sektoren
    move.l  #880*512,44(a1) ;Offset: 880 Sektoren (Root)
    move.l  execbase,a6    ;EXEC-Basisadresse
    jsr     DoIo(a6)       ;Sektoren lesen!

    move.l  diskio+32,d6    ;IO_ACTUAL in D6

```

```
    lea    diskio,a1
    move   #9,28(a1)      ;Command: TD_MOTOR
    move.l  #0,36(a1)      ;Motor aus-
    jsr    DoIo(a6)        ;schalten!

    lea    readreply,a1
    jsr    RemPort(a6)     ;Port entfernen

    lea    diskio,a1
    jsr    closedev(a6)    ;Trackdisk-Device schließen

error:
    rts                    ;Ende

trddevice:  dc.b 'trackdisk.device',0
even

diskio:     blk.l 20,0
readreply:  blk.l 8,0
diskbuff:   blk.b 512*2,0
```

In diesem Beispiel werden die Sektoren 880 und 881 von Laufwerk 0 in den Speicher ab 'diskbuff' geladen. Bei dem Sektor 880 handelt es sich um den Root-Block, der ja den Diskettennamen und einiges mehr enthält.

Danach wird noch einmal DoIo() aufgerufen, um den Laufwerksmotor wieder auszuschalten (für's Einschalten hätte eine 1 in 36(A1) geschrieben werden müssen).

Zum Ablauf des Programmes:

Mit dem Aufruf der FindTask()-Funktion, der als Argument in A1 eine Null übergeben wird, wird der Zeiger auf die eigene Task-Struktur ermittelt. Dieser Zeiger wird dann in die Port-Struktur eingetragen, damit das System weiß, welcher Task nach Beenden der I/O wieder 'aufgeweckt' werden soll.

Als nächstes wird der so vorbereitete Port im System installiert.

Nun wird die Trackdisk-Device geöffnet. In D0 können Sie hierbei wählen, auf welches Laufwerk sich diese Funktion be-

ziehen soll. Wollen Sie mehrere Laufwerke gleichzeitig bearbeiten, so müssen Sie auch mehrere I/O-Strukturen vorbereiten und `OpenDevice()`-Aufrufe machen.

Tritt beim Öffnen der Device ein Fehler auf, so wird zum Label 'error' verzweigt, wo das Programm beendet wird. Andernfalls wird nun die I/O-Struktur mit den notwendigen Daten versehen:

- dem Zeiger auf die Port-Struktur, über die die OK-Meldung der Device erfolgen soll,
- das Kommando, das bei der nächsten I/O-Operation ausgeführt werden soll (hier: `2=CMD_READ`),
- einem Zeiger auf den zu füllenden Pufferspeicher,
- der Länge dieses Speichers und
- für das Lesen von Sektoren die Angabe des Offsets vom Anfang der Diskette, was der Sektor- bzw. Blocknummer*512 entspricht.

Die so vorbereitete I/O-Struktur wird nun mit `DoIo()` dem System übergeben und die gewählte Funktion ausgelöst. Das Programm wartet dann solange, bis die I/O-Operation beendet ist. Ein eventueller Rückgabeparameter, wie etwa beim `TD_PROTSTATUS`-Kommando, wird dann in `IO_ACTUAL` übergeben und mit dem folgenden `MOVE.L`-Befehl ins Datenregister D6 geladen. In obigem Beispiel ist dies der Wert \$400, was der Anzahl der gelesenen Bytes entspricht. Das Datenregister wird zwar nicht weiterverwendet, kann aber bei Verwendung eines Debuggers ausgegeben und abgelesen werden.

Anschließend folgt ein weiterer Aufruf der `DoIo()`-Funktion, diesmal mit dem Kommando `TD_MOTOR` (9). Der Parameter in `IO_LENGTH` (`diskio+36`) gibt dabei an, ob der Motor ein- (1) oder ausgeschaltet (0) werden soll.

Ist dies erledigt, so wird der Port abgemeldet und die Device geschlossen. Das war's.

Sollten Sie dieses Programm im Profimat oder K-SEKA gestartet haben, so werden nach dessen Beendigung die Prozessor-Register ausgegeben. In D6 finden Sie dann den Rückgabeparameter \$400. Sie können nun außerdem durch die Eingabe von 'q diskio+432' (SEKA) den Diskettennamen ausgeben lassen, wobei das erste Byte die Länge des Namens angibt.

Sollte eine Funktion nicht so recht klappen, so wird ein Statuswert im Byte `IO_ERROR` übergeben. Die möglichen Werte sind hierbei:

20	NotSpecified	unbekannter Fehler
21	NoSecHdr	kein Sektor-Header vorhanden
22	BadSecPreamble	ungültiger Sektor-Vorspann
23	BadSecID	ungültige Sektor-ID
24	BadHdrSum	falsche Header-Checksumme
25	BadSecSum	falsche Sektor-Checksumme
26	TooFewSecs	nicht genug Sektoren verfügbar
27	BadSecHdr	ungültiger Sektor-Header
28	WriteProt	Diskette schreibgeschützt
29	DiskChanged	Diskette ist gewechselt worden
30	SeekError	Track nicht gefunden
31	NoMem	nicht genug Speicher
32	BadUnitNum	ungültige Sektornummer
33	BadDriveType	ungültiger Laufwerkstyp
34	DriveInUse	Laufwerk bereits aktiv
35	PostReset	Reset-Phase

Dies war die eine Richtung. Die andere, also das Schreiben von Daten auf die Diskette, läuft genauso ab, nur daß das Kommando auf 3 (`CMD_WRITE`) geändert werden muß. Probieren Sie dies aber bitte nur auf weniger wichtigen Disketten aus, da bei einem Fehler eventuell Daten verlorengehen können.

Ein anderes Kommando ist recht interessant: `TD_FORMAT`. Mit diesem Kommando können ein oder mehrere Tracks auf der Diskette beschrieben werden. Die Daten, die im Speicher bereitliegen müssen und auf die der `IO_DATA`-Zeiger zeigt, werden dann in jeden der angegebenen Tracks geschrieben, ohne daß irgendein Test auf Diskettenwechsel o.ä. stattfindet. Man kann

also mit diesem Kommando nicht nur Disketten formatieren, sondern auch kopieren, indem man aus der einen Diskette die Sektoren liest und mit `TD_FORMAT` diese Daten auf die Zieldiskette zurückschreibt. Der Vorteil dieser Methode gegenüber dem Zurückschreiben mit `CMD_WRITE` ist einfach der, daß die Zieldiskette nicht vorformatiert werden muß!

Wenn Sie allerdings eine ganze Diskette lediglich neu formatieren und nicht das `FORMAT`-Kommando des CLI verwenden wollen, sollten Sie eines bedenken. Die Daten für die Tracks müssen in einem bestimmten Format vorbereitet sein (s. Disketten-Kapitel), was recht viel Arbeit bedeutet. Der Weg über das `TD_FORMAT`-Kommando der Device ist daher so mühsam, daß dies nicht sinnvoll erscheint.

Wir wollen uns nun eine Anwendung der Trackdisk-Device ansehen, bei der auch die Kenntnisse der Disketten-Aufteilung angewandt werden können. Das gleich vorgestellte Maschinenprogramm kann als Diagnose-Programm verwendet werden, entweder aus Neugierde oder zum Feststellen, welche Dateien bei einem Diskettenfehler verloren gehen oder gingen.

Das Programm wird vom CLI aus aufgerufen, wobei als Parameter ein Dateiname mit angegeben werden muß. Es errechnet aus diesem Namen die Hash-Nummer (wie versprochen) und gibt diese aus. Danach lädt es den Root-Sektor der Diskette und gibt den Diskettennamen aus. Mit Hilfe der errechneten Hash-Nummer wird nun die Hash-Kette nach dem angegebenen Namen durchsucht. Wird er nicht gefunden bzw. ist der Eintrag in der Hash-Tabelle unbelegt, so wird '-unknown-' ausgegeben und abgebrochen.

Wird der passende File- bzw. Directory-Header gefunden, so wird dessen Block-Nummer sowie die Anzahl der durch diese Datei belegten Datenblocks ausgegeben.

Es werden dann alle diese Datenblocks der Reihe nach geladen und ihre Nummern ausgegeben. Es wäre zwar auch möglich, diese Blocknummern dem File-Header-Block und evtl. dessen Extension-Block zu entnehmen, jedoch wäre dadurch nicht die

Sicherheit gegeben, daß diese Blocks auch in Ordnung sind. Sollte also einmal der Requester mit der Mitteilung 'Disk-Structure Corrupt' auftauchen, so können Sie mit diesem Programm Ihre wichtigen Dateien auf Vollständigkeit testen.

Um die Übersichtlichkeit des Programmes zu erhalten, ist es allerdings nicht möglich, Dateien aus Unterdirectories zu testen. Dies ist natürlich möglich, wenn man das Programm so ändert, daß es die Hash-Tabelle eines Directory-Header-Blocks anstelle des Root-Blocks verwendet.

Hier nun das Programm, in dem einige interessante Funktionen auftauchen, die Sie auch leicht in Ihren eigenen Programmen verwenden können. Das Programm wurde auf dem K-SEKA-Assembler erstellt, eine Anpassung auf einen anderen Assembler ist aber recht einfach.

;***** File-Tracer 6/87 S.D. *****

```
ExecBase = 4      ;EXEC-Basisadresse
FindTask = -294   ;Task-Struktur suchen
AddPort = -354    ;Port erstellen
RemPort = -360    ;Port entfernen
OpenLib = -408    ;Bibliothek öffnen
CloseLib = -414   ;Bibliothek schließen
OpenDev = -444    ;Device öffnen
CloseDev = -450   ;Device schließen
DoIo = -456       ;I/O starten und warten

output = -60      ;Standard-Ausgabe ermitteln
write = -48       ;Daten ausgeben
```

run:

```
move.l a0,compnt
move.l d0,comlen
move.l execbase,a6
lea dosname,a1      ;Name: dos.library
clr.l d0
jsr openlib(a6)     ;DOS öffnen
move.l d0,dosbase
beq nodos
```

```

move.l dosbase,a6
jsr    output(a6)      ;Standard-Ausgabekanal
move.l d0,outbase

sub.l  #1,commen      ;Namenslänge korrigieren
move.l compnt,a0      ;* Hash-Wert errechnen *
move.l commen,d0      ;Hash=Länge
clr.l  d2
move.l d0,d1
subq   #1,d1          ;Zähler=Länge-1

hashloop:
mulu   #13,d0         ;Hash=Hash*13
move.b (a0)+,d2
bsr    upper          ;umwandeln in Upper-Case
add    d2,d0          ;Hash=Hash+Zeichen
and    #$7ff,d0       ;AND $7FF
dbra   d1,hashloop    ;Schleife

divu   #72,d0
swap   d0             ;Hash modulo 72
addq   #6,d0          ;+6
move   d0,hash        ;Hash errechnet!

move.l #hashtxt,d2
bsr    prtxt          ;"Hash:" ausgeben
move   hash,d0
bsr    phex           ;Hash-Nummer ausgeben

move.l execbase,a6    ;Zeiger auf EXEC-Bibliothek
sub.l  a1,a1          ;eigener Task
jsr    FindTask(a6)   ;Task suchen
move.l d0,readreply+$10 ;set SigTask

lea    readreply,a1
jsr    AddPort(a6)    ;Add Reply-Port

lea    diskio,a1
clr.l  d0
clr.l  d1
lea    trdddevice,a0
jsr    OpenDev(a6)    ;Open trackdisk.device
tst.l  d0
bne    error

```

```

move.l #880,d0          ;Sektor 880 (Root-Sektor)
bsr     loadsec          ;in Diskpuffer laden

move.l #voltxt,d2
bsr     prttxt           ;"Volume:" ausgeben
move.l dosbase,a6
move.l outbase,d1
move.l #diskbuff+433,d2 ;Namensadresse
clr.l   d3
move.b diskbuff+432,d3  ;Namenslänge
jsr     write(a6)        ;Disknamen ausgeben

lea     diskbuff,a0
clr.l   d0
move    hash,d0
lsl     #2,d0            ;Hash*4=Sektor-Zeiger
move.l  0(a0,d0),d0      ;Sektornummer holen
tst.l   d0 ;Zeiger da?
beq     none            ;nein: Hash-Eintrag unbelegt!
loadloop:
move.l  d0,sector
bsr     loadsec          ;nächsten Sektor laden

move.l  compnt,a0
lea     diskbuff+432,a1 ;Namenslänge aus Header
move.l  commlen,d0
cmp.b   (a1)+,d0         ;stimmt die Länge?
bne     nextsec          ;nein

subq    #1,d0
namelop:
move.b  (a1)+,d2
bsr     upper            ;Zeichen in Upper-Case
move    d2,d1
move.b  (a0)+,d2
bsr     upper            ;Zeichen in Upper-Case
cmp.b   d1,d2            ;Zeichen vergleichen
bne     nextsec          ;falsch
dbra    d0,namelop
bra     sectorok         ;Name stimmt!

nextsec:
move.l  diskbuff+496,d0  ;Zeiger auf nächsten Sektor
tst.l   d0               ;ist einer da?
bne     loadloop         ;ja: weiter

```

```

none:                                ;sonst
    move.l    #unknown,d2
    bsr       prtxt                  ;"-unknown-" ausgeben
    bra       ende                  ;und Ende

sectorok:
    move.l    #header,d2
    bsr       prtxt                  ;"Header:" ausgeben
    move.l    sector,d0              ;Sektor-Header-Nummer
    bsr       phex                   ;ausgeben

    cmp.l     #2,diskbuff+508        ;Dir-Header?
    bne       nodir                 ;nein
    move.l    #dirtxt,d2
    bsr       prtxt                  ;"Directory" ausgeben
    bra       ende                  ;und Ende

nodir:
    move.l    diskbuff+504,d0        ;Extension
    tst.l     d0                    ;existent?
    beq       noextens              ;nein
    move.l    d0,-(sp)               ;D0 retten
    move.l    #extxt,d2
    bsr       prtxt                  ;"Extension" ausgeben
    move.l    (sp)+,d0               ;Sektor-Nr. holen
    bsr       phex                   ;und ausgeben

noextens:
    move.l    #crtxt,d2
    bsr       prtxt                  ;CR ausgeben
    move.l    diskbuff+8,d0
    bsr       phex                   ;Sektor-Anzahl ausgeben
    move.l    #sectxt,d2
    bsr       prtxt                  ;"Sektoren" ausgeben
    clr       counter                ;Spaltenzähler=0
    bra       seclop1               ;Sektoren ausgeben

secloop:
    move.l    sector,d0
    bsr       phex                   ;Sektornummer ausgeben
    add       #1,counter              ;Zähler+1
    cmp       #8,counter             ;8 Zahlen ausgegeben?
    bne       seclop1               ;nein
    clr       counter                ;sonst Zähler löschen
    move.l    #crtxt,d2
    bsr       prtxt                  ;und CR ausgeben

```

```

secloop1:
    move.l    diskbuff+16,d0    ;nächster Sektor
    tst.l     d0                ;vorhanden?
    beq       ende              ;nein: fertig
    move.l    d0,sector
    bsr       loadsec            ;nächsten Sektor laden
    bra       secloop           ;usw...

ende:
    move.l    #crtxt,d2
    bsr       prttxt             ;CR ausgeben

    move.l    execbase,a6
    lea       readreply,a1
    jsr       RemPort(a6)        ;Remove Port
    lea       diskio,a1
    jsr       closedev(a6)       ;Close Trackdisk-Device
error:
    move.l    dosbase,a1
    jsr       closelib(a6)       ;Close DOS
nodos:
    rts                          ;Ende

loadsec:
                                ;Sektor D0 laden
    lea       diskio,a1
    move.l    #readreply,14(a1) ;set Reply-Port
    move      #2,28(a1)          ;Command: READ
    move.l    #diskbuff,40(a1) ;Puffer
    move.l    #512,36(a1)       ;Länge: 1 Sektor
    mulu      #512,d0
    move.l    d0,44(a1)         ;Offset: Sektornummer+512
    move.l    execbase,a6
    jsr       DoIo(a6)          ;Sektor lesen

    lea       diskio,a1
    move      #9,28(a1)          ;TD_MOTOR
    move.l    #0,36(a1)         ;Motor aus
    jsr       DoIo(a6)
    rts

phex:
                                ;D0 sedezimal ausgeben
    lea       outpuff,a0
    move      d0,d2
    move      #3,d3              ;4 Ziffern
niblop:
    rol       #4,d2              ;linkes Nibble nach unten

```

```

    move    d2,d1
    and     #$f,d1          ;ausmaskieren
    add     #$30,d1         ;zu ASCII machen
    cmp     #'9',d1         ;Ziffer?
    bls     nibok           ;ja
    add     #7,d1           ;sonst korrigieren
nibok:
    move.b  d1,(a0)+        ;Zeichen in Ausgabepuffer
    dbra    d3,niblop       ;Schleife fortführen
    move.b  #$20,(a0)       ;Leerzeichen ans Ende

    move.l  dosbase,a6
    move.l  outbase,d1
    move.l  #outpuff,d2     ;Ausgabepuffer
    move.l  #5,d3           ;5 Zeichen
    jmp     Write(a6)       ;ausgeben

prtxt:
                                ;Text ab (D2) ausgeben
    move.l  dosbase,a6
    move.l  outbase,d1
    move.l  #12,d3          ;12 Zeichen Länge
    jmp     Write(a6)       ;Text ausgeben

upper:
                                ;D2 umwandeln in Upper-Case
    cmp.b  #'a',d2          ;Zeichen <'a'?
    blo     upperx          ;ja: so lassen
    cmp.b  #'z',d2          ;Zeichen >'z'?
    bhi     upperx          ;ja: so lassen
    sub     #$20,d2         ;sonst korrigieren
upperx:
    rts                    ;fertig

trddevice:  dc.b 'trackdisk.device',0
dosname:    dc.b 'dos.library',0

hashtxt:    dc.b $a,'Hashnum:  '
voltxt:     dc.b $a,'Volume:   '
unknown:    dc.b $a,'-unknown- '
header:     dc.b $a,'Header:   '
extxt:      dc.b $a,'Extension: '
dirtxt:     dc.b $a,'Directory ', $a
sectxt:     dc.b 'Sektoren:  ', $a
crtxt:      dc.b '            ', $a

```


data

even

```
outpuff:    blk.b 6      ;Puffer für Hexzahlen-Ausgabe
sector:     blk.l 1      ;Sektor-Zwischenspeicher
counter:    blk.w 1      ;Zähler für Ausgabeformatierung
dosbase:    blk.l 1      ;DOS-Basisadresse
outbase:    blk.l 1      ;Standard-Ausgabe-Handle
hash:       blk.w 1      ;Hash-Nummer
commpnt:    blk.l 1      ;Zeiger auf Eingabezeile
commllen:   blk.l 1      ;Länge der Eingabezeile
```

```
diskio:                ;Disk-I/O-Struktur
```

```
message:    blk.b 20,0
```

```
io:         blk.b 12,0
```

```
ioreq:      blk.b 16,0
```

```
readreply:  blk.l 8,0
```

```
diskbuff:   blk.b 512,0
```

Auf diese Weise läßt sich auch ein Programm schreiben, das eine Datei von der Diskette lädt, ohne dabei das DOS zu bemühen. Es müssen dann nur die eigentlichen Daten aus den Datenblocks in den Speicher kopiert werden.

4.2 Consol-Device: Editor-Fenster

Diese Device, mit der Fenster für Tastatur-Ein- und -Ausgaben vorbereitet und bearbeitet werden können, fällt ein wenig aus dem Rahmen der Standard-Devices. Diese Device kann nämlich nicht so einfach für sich geöffnet und verwendet werden, sondern muß in Verbindung mit einem Fenster stehen. Dieses Fenster dient dann für die Ein- und Ausgaben der Console-Device.

Bevor also die Device selbst geöffnet werden kann, müssen wir erst einmal ein Fenster öffnen. Dafür müssen wir die Intuition-Bibliothek, einen Screen und dann das Fenster öffnen. Den dadurch erhaltenen Zeiger auf die Fenster-Struktur übergeben wir dann beim Öffnen der Console-Device.

Wir erhalten somit also ein Fenster auf einem eigenen Screen, in dem ein Cursor in der linken oberen Ecke zu sehen ist. Dieser Cursor hat allerdings noch keine Funktion; wir müssen die Tastatur-Eingabe und die Ausgabe der Zeichen in das Fenster erst einmal programmieren.

Wir benötigen also zwei I/O-Strukturen, eine für die Eingaben und eine für die Ausgaben. Dazu gehören natürlich auch zwei Message-Ports, damit die Device auch erfährt, wohin bzw. woher die Daten kommen sollen.

Bevor wir nun mit der doch recht trockenen Theorie fortfahren, sollten Sie sich erst einmal das folgende Programm ansehen, das die oben beschriebenen Schritte vornimmt. Es öffnet einen Screen und ein Fenster, in dem dann über die Console-Device Ein- und Ausgaben laufen. Dabei werden lediglich die über die Tastatur eingegebenen Zeichen im Fenster wieder ausgegeben, wobei 'Return' und 'Backspace' speziell behandelt werden. Wird die Close-Box des Fensters mit der Maus angeklickt, so wird das Programm beendet. Weitere Aktionen mit der Maus können zusätzlich ausgewertet werden, die entsprechende Erkennung eines Maus-Klicks ist vorbereitet.

Hier das Programm:

;** Demo-Programm zur Console-Device 6/87 S.D. **

openlib	= -408	;Library öffnen
closelib	= -414	;Library schließen
AddPort	= -354	;Port erstellen
RemPort	= -360	;Port entfernen
OpenDev	= -444	;Device öffnen
CloseDev	= -450	;Device schließen
execbase	= 4	;EXEC-Basisadresse
GetMsg	= -372	;Message holen
FindTask	= -294	;Task ermitteln
DoIo	= -456	;I/O ausführen
SendIo	= -462	;I/O starten

```

;    ** Intuition-Funktionen **
openscreen  = -198      ;Screen öffnen
closescreen = -66       ;Screen schließen
openwindow  = -204      ;Fenster öffnen
closewindow = -72       ;Fenster schließen

run:
    bsr    openint      ;Intuition öffnen
    bsr    scropen      ;Screen öffnen
    bsr    windopen     ;Fenster öffnen

    move.l  execbase,a6    ;Zeiger auf EXEC-Bibliothek

    sub.l   a1,a1          ;eigener Task
    jsr    FindTask(a6)    ;Task suchen
    move.l  d0,readreply+$10 ;SigTask setzen

    lea     readreply,a1
    jsr     AddPort(a6)    ;Add Read-Reply-Port

    lea     writerep,a1
    jsr     AddPort(a6)    ;Add Write-Reply-Port

    lea     readio,a1
    move.l  windowhd,readio+$28 ;unser Window
    move.l  #48,readio+$24 ;Länge der Struktur
    clr.l   d0
    clr.l   d1
    lea     devicename,a0
    jsr     OpenDev(a6)    ;Open Consol-Device
    tst.l   d0
    bne     error

    move.l  readio+$14,writeio+$14 ;DEVICE und
    move.l  readio+$18,writeio+$18 ;UNIT kopieren

go:
    bsr     queueread      ;Eingabe anwerfen

loop:
                                ;* Ereignisse auswerten *
    move.l  execbase,a6
    move.l  windowhd,a0
    move.l  86(a0),a0      ;Fenster-User-Port
    jsr     GetMsg(a6)
    tst.l   d0

```

```

    bne    wevent          ;Fenster-Ereignis

    lea    readreply,a0
    jsr    GetMsg(a6)      ;Console-Ereignis (Taste)?
    tst.l  d0
    beq    loop            ;kein Ereignis

cevent:                                ;* Taste verarbeiten *
    bsr    conout          ;Zeichen ausgeben
    cmp.b  #$d,buffer      ;Return?
    bne    no1             ;nein
    move.b  #$a,buffer     ;sonst LF ausgeben
    bsr    conout

no1:
    cmp.b  #$8,buffer      ;Backspace?
    bne    no2
    move.b  #' ',buffer    ;sonst Zeichen löschen
    bsr    conout
    move.b  #8,buffer      ;und wieder zurück
    bsr    conout

no2:
    bra    go              ;und so weiter

wevent:                                ;* Fenster-Ereignis auswerten *
    move.l  d0,a0
    move.l  $16(a0),d6      ;Message in D6
    cmp.l  #$2000000,d6     ;Window-Close?
    beq    ende            ;ja: Ende

    ;* hier kann eine weitere Auswertung stattfinden: *

    move.l  windowhd,a0
    move.l  12(a0),d5       ;Mausposition in D5

    ;* z.B. Cursor setzen auf Maus-Position... *

ende:                                ;* Programmende: alles schließen *
    lea    readreply,a1
    jsr    RemPort(a6)      ;Remove Port
    lea    readio,a1
    jsr    closedev(a6)     ;Close Device
    lea    writerep,a1
    jsr    RemPort(a6)      ;Remove Port

```

```
error:
    bsr    windclose      ;Fenster schließen
    bsr    scrclose      ;Screen schließen
    bsr    closeint      ;Intuition schließen

    rts                    ;* ENDE *

; ** Unter-Routinen **

queueread:                    ;* Consol-Eingabe starten *
    move.l    execbase,a6
    lea      readio,a1
    move      #2,28(a1)      ;Command: READ
    move.l    #buffer,40(a1) ;Puffer
    move.l    #1,36(a1)      ;Länge:
    move.l    #readreply,14(a1) ;set Reply-Port
    jsr      sendio(a6)      ;Funktion auslösen
    rts

conout:                      ;* 1 Zeichen ausgeben *
    move.l    execbase,a6
    lea      writeio,a1
    move      #3,28(a1)      ;Command: WRITE
    move.l    #buffer,40(a1) ;Puffer
    move.l    #1,36(a1)      ;Länge:
    move.l    #writerep,14(a1) ;set Reply-Port
    jsr      DoIo(a6)        ;Funktion ausführen
    rts

openint:                    ;* Intuition öffnen *
    move.l    execbase,a6
    lea      intname,a1      ;Library-Name
    jsr      openlib(a6)
    move.l    d0,intbase
    rts

closeint:                    ;* Intuition schließen *
    move.l    execbase,a6
    move.l    intbase,a1
    jsr      closelib(a6)
    rts

scropen:                    ;* Screen öffnen *
    move.l    intbase,a6
    lea      screen_defs,a0
```

```
    jsr    openscreen(a6)
    move.l  d0,screenhd
    rts

scrclse:                ;* Screen schließen *
    move.l  intbase,a6
    move.l  screenhd,a0
    jsr    closescreen(a6)
    rts

windopen:               ;* Fenster öffnen *
    move.l  intbase,a6
    lea     windowdef,a0
    jsr    openwindow(a6)
    move.l  d0>windowhd
    rts

windclose:              ;* Fenster schließen *
    move.l  intbase,a6
    move.l  windowhd,a0
    jsr    closewindow(a6)
    rts

screen_defs:            ;* Screen-Struktur *
    dc.w 0,0            ;Position
    dc.w 640,200        ;Größe
    dc.w 4              ;Bitmaps
    dc.b 0,1            ;Farben
    dc.w $800           ;Modus
    dc.w 15             ;Typ
    dc.l 0              ;Standard-Zeichensatz
    dc.l titel          ;Screen-Titel
    dc.l 0              ;Standard-Titel
    dc.l 0              ;keine Gadgets

windowdef:              ;* Fenster-Struktur *
    dc.w 10,20          ;Position
    dc.w 300,150        ;Größe
    dc.b 0,1            ;Farben
    dc.l $208           ;IDCMP-Flags
    dc.l $100f          ;Window-Flags
    dc.l 0              ;keine Gadgets
    dc.l 0              ;keine Menu-Checks
    dc.l windname       ;Fenstername
```

```
screenhd: dc.l 0      ;Screen-Struktur-Zeiger
          dc.l 0      ;keine Bitmap
          dc.w 100,50  ;Mindestgröße
          dc.w 300,200 ;Höchstgröße
          dc.w $f      ;Screen-Typ
```

```
titel:      dc.b "Editor-Screen",0
windname:   dc.b "Console-Fenster",0
iname:      dc.b "intuition.library",0
devicename: dc.b 'console.device',0
even
windowhd:   blk.l 1
intbase:    blk.l 1
conbase:    blk.l 1
```

```
readio:
message:    blk.b 20,0
io:         blk.b 12,0
ioreq:      blk.b 16,0
```

```
writeio:
          blk.b 20,0
          blk.b 12,0
          blk.b 16,0
```

```
readreply:  blk.l 8,0
writerep:   blk.l 8,0
buffer:     blk.b 80,0
```

Die Sequenzen, die diverse Funktionen im Fenster auslösen können, sind dieselben wie bei den mittels des DOS geöffneten RAW:- bzw. CON:-Fenstern.

4.3 Narrator-Device: Sprachausgabe

Der Narrator, was auf deutsch soviel wie Erzähler heißt, ermöglicht dem Amiga, sich verbal auszudrücken, sprich zu reden. Dies ist sicher ein sehr interessanter Punkt, mit dem man seine Programme buchstäblich ansprechender gestalten kann.

Der Narrator ist ein Programmpaket, das als Device aufgebaut ist. Man kann somit auch einen Text sprachlich ausgeben lassen und währenddessen weiterarbeiten. Die Extended-I/O-Struktur der Narrator-Device ist folgendermaßen aufgebaut:

Wort	Name	Bedeutung
0	RATE	Sprechgeschwindigkeit in Worte/Minute
1	PITCH	Sprachgrundfrequenz in Hertz
2	MODE	Sprachmodus (0=mit, 1=ohne Betonung)
3	SEX	Stimm-Geschlecht (0=männlich, 1=weiblich)
4	CHMASKS	Zeiger auf Kanal-Maskenfeld
6	NUMMASKS	Anzahl der Kanal-Masken
7	VOLUME	Lautstärke
8	SAMPFREQ	Abtastrate
9	MOUTHS	Munderzeugungs-Flag (Byte)
	CHANMASK	aktueller Kanal (nur interne Bedeutung)

Die Programmierung der Narrator-Device ist so ähnlich wie die der anderen Devices. Was jedoch dazukommt, ist die Verwendung des Translators, der einen normalen Text in die Lautschrift für den Narrator übersetzt. Dieser Translator ist keine Device, sondern eine Bibliothek, die nur eine einzige Funktion enthält.

Hier ein Maschinen-Programm, das einen Beispieltext ausgibt. Mit diesem Programm können Sie schön herumexperimentieren, da Sie die Parameter beliebig ändern und das Ergebnis testen können. Vorteilhaft ist hier ebenfalls die Verwendung eines

Assemblers mit eingebauten Debuggers wie dem Profimat oder dem K-SEKA, auf dem dieses Programm geschrieben wurde.

;***** Narrator-Demo 6/87 S.D. *****

```
ExecBase=4      ;EXEC-Basisadresse
FindTask =-294  ;Find Task
AddPort =-354   ;Add Port
RemPort =-360   ;Remove Port
OpenLib=-408    ;Open Library
closelib=-414   ;Close Library
OpenDev =-444   ;Open Device
CloseDev =-450  ;Close Device
DoIo =-456      ;Do I/O
SendIo = -462   ;Send I/O
Translate =-30  ;Translate Text
```

run: ;** System initialisieren und öffnen **

```
move.l  execbase,a6
lea     transname,a1
clr.l   d0
jsr     openlib(a6)      ;Open Translator-Library
move.l  d0,tranbase
beq     error

sub.l   a1,a1            ;Task-Nummer = 0: eigener Task
jsr     FindTask(a6)     ;Find Task

move.l  d0,writerep+$10 ;set SigTask

lea     writerep,a1
jsr     addport(a6)      ;Add Reply-Port

lea     talkio,a1
clr.l   d0
clr.l   d1
lea     nardevice,a0
jsr     opendev(a6)      ;Open Narrator.device
tst.l   d0
bne     error

lea     talkio,a1
move.l  #writerep,14(a1) ;set Reply-Port (*)
move    #150,48(a1)      ;Rate (40-400)
move    #110,50(a1)      ;Pitch (65-320)
move    #0,52(a1)        ;Mode: mit Betonung (0/1)
```

```

move    #0,54(a1)      ;Geschlecht: männlich (0/1)
move.l   #amaps,56(a1)  ;Masks (*)
move     #4,60(a1)      ;4 Masken (*)
move     #64,62(a1)     ;Lautstärke (0-64)
move     #22200,64(a1)  ;Abtastrate (5000-28000)

sayit:                                ;** Text übersetzen und sagen **
    lea    intext,a0      ;Original-Text
    move.l #outtext-intext,d0 ;dessen Länge
    lea    outtext,a1     ;Puffer für Übersetzung
    move.l #512,d1        ;dessen Länge
    move.l tranbase,a6
    jsr    Translate(a6)  ;Text übersetzen

    lea    talkio,a1
    move    #3,28(a1)     ;Command: Write
    move.l  #512,36(a1)   ;Length
    move.l  #outtext,40(a1) ;Buffer
    move.l  execbase,a6
    jsr    DoIo(a6)       ;Say it!!

qu:                                ;** Ende **
    lea    writerep,a1
    jsr    RemPort(a6)    ;Remove Port

    lea    talkio,a1
    jsr    closedev(a6)   ;Close Narrator

    move.l tranbase,a1
    jsr    closelib(a6)   ;Close Translator-Lib

    clr.l  d0
error:
    rts

transname: dc.b 'translator.library',0
nardevice: dc.b 'narrator.device',0
amaps:     dc.b 3,5,10,12
intext:    dc.b 'hello, i am the amiga computer.',0
even
outtext:   blk.l 128,0
tranbase:  blk.l 1,0
narread:   blk.l 20,0
talkio:    blk.l 20,0
writerep:  blk.l 8,0

```

In der Vorbereitung des I/O-Bereiches für die verschiedenen Modi bzw. Raten sind nur die Werte unbedingt einzutragen, die im obigen Listing mit einem Sternchen (*) versehen sind. Alle anderen werden automatisch auf die Standardwerte gesetzt (Default). Diese Werte sind auch im Programm vorgegeben, sie können in den in Klammern angegebenen Grenzen variiert werden.

Die Narrator-Device bietet zusätzlich die Möglichkeit, während der Sprachausgabe Daten an das aufrufende Programm zu übertragen. Dies ist natürlich nur dann möglich, wenn die Sprachausgabe nicht mit DoIo(), sondern mit SendIo() gestartet wird, damit auch während der laufenden Sprachausgabe die Daten empfangen werden können.

Die so empfangenen Daten stellen ein Bitmuster dar, das auf dem Bildschirm ausgegeben einen Mund darstellt, der den gerade gesprochenen Lauten entspricht. Diese Möglichkeit soll hier aber nicht näher erläutert werden, da dies mit Graphikausgaben verbunden ist, die ja in ein Graphikbuch gehören. Es sei nur erwähnt, daß diese Graphik recht primitiv ist, da sie den sprechenden Mund nur als entsprechend hohes Parallelogramm darstellt. Die Breite und Höhe dieser Form bekommt man von der Device in der Erweiterung der Read-Request-I/O-Struktur.

Diese Erweiterung hat folgenden Aufbau:

Offset	Name	Inhalt
48	MRB_WIDTH	Breite der 'Mundform'
49	MRB_HEIGHT	Höhe
50	MRB_SHAPE	internes Datenbyte
51	MRB_PAD	Füllbyte für gerade Adresse

Die I/O-Funktion kann natürlich auch schiefgehen. Die sich dann ergebenden Fehlermeldungen können folgende Werte annehmen:

Narrator-Fehlermeldungen:

Nummer	Bedeutung
-2	nicht genug Speicher da
-3	Audio-Device nicht vorhanden
-4	Library nicht erstellbar
-5	falsche Unit-Nummer in der I/O-Struktur (nur 0)
-6	keine Audio-Kanäle verfügbar
-7	unbekanntes Kommando
-8	Mund-Daten gelesen, aber nicht geschrieben
-9	kein Öffnen möglich
-20	Lautschrift unaussprechlich
-21	ungültige Rate
-22	ungültiges Pitch
-23	ungültiges Geschlecht
-24	ungültiger Modus
-25	ungültige Abtastrate
-26	ungültige Lautstärke

4.4 Serial-Device: die RS232-Schnittstelle

Diese Device ist für die serielle Kommunikation mit der Außenwelt zuständig. Auch diese Ein- und Ausgaben über den seriellen Port des Amiga lassen sich mit normalen DOS-Funktionen realisieren, wenn man als Dateinamen SER: angibt. Diese Methode hat jedoch einige große Nachteile.

Der übliche Nachteil der DOS-Verwendung ist bekanntlich der, daß die Ein-/Ausgaben nicht im Hintergrund laufen und man somit auf deren Beendigung warten muß. Dies läßt sich bei Device-Programmierung mittels der `SendIo()`-Funktion vermeiden.

Ein weiterer Nachteil, der gerade in diesem Fall auftritt, ist der, daß man die seriellen Parameter wie z.B. die Übertragungsrate vorher mit dem Preferences-Programm eingestellt haben muß.

Die Serial-Device bietet hierfür eine eigene Funktion, mit der alle Parameter eingestellt werden können. Für diese und die anderen Funktionen steht wieder eine erweiterte I/O-Struktur zur Verfügung, die folgende Einträge besitzt (Standard-Werte in Klammern):

Offset	IO_Name	Inhalt
0	CTLCHAR	Kontrollzeichen: xON, xOFF, frei, frei (\$11130000)
4	RBUFLLEN	Eingabepuffer-Länge (\$200)
8	WBUFLLEN	Ausgabepuffer-Länge (\$200)
12	BAUD	Baudrate (9600)
16	BRKTIME	Break-Länge in Mikrosekunden (250000)
20	TERMARRAY	Abbruchzeichenfeld (8 Bytes)
28	READLEN	Bits pro Zeichen beim Lesen (8)
29	WRITELEN	Bits pro Zeichen beim Senden (8)
30	STOPBITS	Anzahl der Stoppbits (1)
31	SERFLAGS	Seriell-Flags (s.u.) (\$20)
32	STATUS	Statuswort

Die Bits des zurückgegebenen Statuswortes sind folgende:

Bit	wenn	dann
0	0	busy, Übertragung läuft
1	0	paper out, Empfänger nicht bereit
2	0	select, angewählt
3	0	Data Set Ready (DSR)
4	0	Clear To Send (CTS)
5	0	Carrier Detect (CD)
6	0	Ready To Send (RTS)
7	0	Data Terminal Ready (DTR)
8	1	read overrun, Puffer-Überlauf
9	1	break sent, Break gesendet
10	1	break received, Break empfangen
11	1	transmit x-OFFed, xOFF gesendet
12	1	receive x-OFFed, xOFF empfangen
13-15		reserviert

Die Bits des Flags in IO_SERFLAGS haben folgende Bedeutungen:

Bit	Name	Bedeutung, wenn gesetzt
0	PARITY_ON	Parity-Bit gewünscht
1	PARITY_ODD	ungerade Parity
2		unbenutzt
3	QUEUEDBRK	Break im Hintergrund
4	RAD_BOOGIE	Hochgeschwindigkeits-Modus ein
5	SHARED	allgemeiner Zugriff ermöglicht
6	EOFMODE	EOF-Erkennung eingeschaltet
7	XDISABLED	xON/xOFF ausgeschaltet

Die Serial-Device besitzt außer den Standard- drei zusätzliche Kommandos:

Zahl	SDCMD_Name	Funktion
9	QUERY	
10	BREAK	Break senden
11	SEPARAMS	Parameter einstellen

Um vor allem das letzte dieser Kommandos, SDCMD_SETPARAMS, zu demonstrieren, folgt nun wieder ein Beispielprogramm. In diesem Programm wird die Baudrate auf 1200 eingestellt und dann der berühmte Text 'Hello, world!' gesendet.

```
;***** Serial-Device-Demonstration 6/87 S.D. *****
```

```
ExecBase = 4      ;EXEC-Basisadresse
FindTask = -294   ;Task-Struktur suchen
AddPort = -354    ;Port erstellen
RemPort = -360    ;Port entfernen
OpenLib = -408    ;Bibliothek öffnen
CloseLib = -414   ;Bibliothek schließen
OpenDev = -444    ;Device öffnen
CloseDev = -450   ;Device schließen
DoIo = -456       ;I/O starten und warten
```

```

output    = -60      ;Standard-Ausgabe ermitteln
write     = -48      ;Daten ausgeben

run:
    move.l  execbase,a6      ;Zeiger auf EXEC-Bibliothek
    sub.l   a1,a1           ;eigener Task
    jsr     FindTask(a6)     ;Task suchen
    move.l  d0,reply+$10     ;SigTask setzen

    lea     reply,a1
    jsr     AddPort(a6)      ;Reply-Port erstellen

    lea     devio,a1        ;Zeiger auf I/O-Struktur

    clr.l   d0
    clr.l   d1
    lea     devicename,a0
    jsr     OpenDev(a6)      ;Serial-Device öffnen
    tst.l   d0               ;OK?
    bne     error           ;nein: Ende

    move.l  #reply,14(a1)    ;set Reply-Port

    move     #11,28(a1)      ;Command: SETPARAMS
    move.l   #1200,ioextd+12 ;1200 Baud
    jsr     DoIo(a6)         ;Parameter setzen

    move     #3,28(a1)       ;Command: WRITE
    move.l   #text,40(a1)    ;Puffer
    move.l   #textl,36(a1)   ;Länge:
    jsr     DoIo(a6)         ;Text senden

    lea     reply,a1
    jsr     RemPort(a6)      ;Remove Port

    lea     devio,a1
    jsr     CloseDev(a6)     ;Close Device

error:
    rts                    ;Ende

devicename: dc.b 'serial.device',0
text:       dc.b 'hello, world!'
textl       = *-text
even

```

```

devio:
message: blk.w 10,0
io:      blk.w  6,0
ioreq:   blk.w  8,0
ioextd:  blk.w 17,0
reply:   blk.w  8,0

```

Auf diese Weise können natürlich auch mehr Parameter der RS232-Schnittstelle eingestellt werden.

4.5 Printer-Device: Drucker-Programmierung

Auch der Drucker läßt sich noch auf andere Weise als über den Kanal PRT: ansprechen. Dafür dient die Printer-Device, die außer dem normalen Druckbetrieb noch eine weitere interessante Fähigkeit besitzt, nämlich die Ausgabe eines Fenster- oder Screen-Inhaltes auf dem Drucker.

Für den normalen Drucker-Betrieb wird folgende I/O-Struktur-Erweiterung mit dem Namen IOPrtCmdReq benötigt:

Offset	Name	Inhalt
32	io_PrtCommand	Druckerbefehl
34	io_Param0	Kommando-Parameter
35	io_Param1	Kommando-Parameter
36	io_Param2	Kommando-Parameter
37	io_Param3	Kommando-Parameter

Hier werden die Steuer-Kommandos an den Drucker übergeben, die Schriftarten etc. einstellen. Dies wird in Verbindung mit dem PRTCOMMAND-Befehl ausgegeben.

Die möglichen Kommandos, die bei dieser Device zu den Standard-Kommandos beigefügt sind, sind folgende:

Wert	Name	Funktion
9	PRD_RAWWRITE	Ausdrucken ohne Konvertierung von Steuerzeichen
10	PRD_PRTCOMMAND	Druckerkommando senden
11	PRD_DUMPRPORT	Ausdruck eines Screen- oder Fenster-Inhaltes

Wird nicht RAWWRITE verwendet, sondern die Daten über die normale WRITE-Funktion ausgegeben, so werden die Amiga-Standard-Steuerzeichen je nach installiertem Drucker in die druckerspezifischen Steuerzeichen umgewandelt. Dadurch kann ein und dasselbe Programm seine Ausgaben, egal welcher Form, auf jeden beliebigen Drucker senden, ohne daß es über die Eigenarten dieses Druckers Bescheid wissen muß.

Das Kommando DUMPRPORT bietet, wie bereits erwähnt, die Möglichkeit, den Inhalt eines Fensters oder Screens auf dem Drucker auszugeben. Die hierfür notwendige Zusatz-I/O-Struktur mit dem Namen IODRPREq ist folgendermaßen aufgebaut:

Offset	Name	Inhalt
32	RastPort	Zeiger auf auszugebenden Rastport
36	ColorMap	Zeiger auf die Farbtabelle
40	Modes	Graphik-Modus des Viewport
44	SrcX	X-Position des Fensters/Screens
46	SrcY	Y-Position
48	SrcWidth	Fenster-/Screen-Breite
50	SrcHeight	-Höhe
52	DestCols	Zielbreite
56	DestRows	Zielhöhe
60	Special	Flags für Sonderfunktionen

4.6 Parallel-Device: Digital-Ein-/Ausgaben

An dem gleichen Anschluß, an dem normalerweise der Drucker hängt, können auch andere Geräte angeschlossen werden, wenn sie über die entsprechenden elektrischen Eigenschaften verfügen. Es lassen sich dann digitale Daten sowohl ein- als auch ausgeben. Dabei hat man auch noch zusätzlich die Möglichkeit, einzelne Bits der 8 Datenleitungen als Eingang und die anderen als Ausgang zu programmieren. Dies ist allerdings nur durch die direkte Programmierung des Hardware-Registers \$BFE301 möglich.

Bleiben wir daher bei der Programmierung des gesamten Ports als Ein- oder Ausgang. Auch hierfür gibt es eine Device: die Parallel-Device. Für die Verwendung dieser Device muß wieder zusätzlich zu der normalen I/O-Struktur eine Erweiterung vorgenommen werden. Diese Erweiterung hat folgenden Aufbau:

Offset	Name	Inhalt
48	PWBufLen	Länge des Ausgabepuffers
52	ParStatus	Status der Device
53	ParFlags	Parallel-Flags
54	PTermArray	Abbruch-Maske
-61		

Das Status-Byte enthält folgende Status-Bits:

Bit	Name	Bedeutung, wenn gesetzt
0	PSEL	Drucker angewählt
1	PAPEROUT	Papier alle
2	PBUSY	Drucker beschäftigt
3	RWDIR	Datenrichtung (0=lesen, 1=schreiben)

Die Parallel-Flags stellen folgende Bits dar:

Bit	Name	Bedeutung, wenn gesetzt
1	EOFMODE	EOF-Modus eingeschaltet
5	SHARED	Zugriff auch für andere Tasks möglich

Wenn aus dem Parallelport gelesen wird, stellt sich wieder einmal die Frage, woran der Empfänger das Ende der Übertragung erkennen soll. Dafür gibt es hier die Möglichkeit, bei einer bestimmten Byte-Sequenz den Empfang abbrechen zu lassen. Diese Sequenz wird in den zwei Langworten des TermArray eingetragen. Aktiviert wird diese Abbruch-Sequenz, wenn Bit 1 des Flags-Bytes gesetzt wird (EOFMODE) und dann das SETPARAMS-Kommando (10) aufgerufen wird.

4.7 Gameport-Device: Maus und Joystick

Mit dieser Device werden alle Eingaben aus den beiden Ports bearbeitet, die von einer Maus oder von einem Joystick kommen. Die I/O-Struktur dieser Device benötigt keine Erweiterung, jedoch werden zwei weitere Strukturen verwendet.

Eine dieser Strukturen ist die Event-Struktur, die bereits im Kapitel über die RAW:-Fenster vorgestellt wurde. Diese Struktur namens InputEvent hat folgenden Aufbau:

Offset	Name	Inhalt
0	NextEvent	evtl. Zeiger auf die folgende Struktur
4	Class	Ereignis-Klasse
5	SubClass	evtl. Unterklasse des Ereignisses
6	Code	Ereignis-Code
8	Qualifier	Ereignis-Typ
10	X	X-Position
12	Y	Y-Position, meist relativ
14	TimeStamp	Sekunden, Mikrosekunden

Die andere Struktur ist für die Einstellung des Ereignisses nötig, das die Übergabe der Parameter in der Event-Struktur auslöst. Dies kann das Drücken oder Loslassen eines Knopfes oder die horizontale bzw. vertikale Bewegung der Maus oder des Joysticks sein. Der gewünschte Wert ist dafür in das entsprechende Wort der folgenden Struktur namens GamePortTrigger einzutragen:

Offset	Name	Inhalt
0	Keys	Tastenänderung: Bit 0: Taste gedrückt Bit 1: Taste losgelassen
2	Timeout	Abbruch bei Ablauf dieser Anzahl von 1/50 Sekunden
4	XDelta	horizontale Bewegung
6	YDelta	vertikale Bewegung

Will man also 10 Sekunden darauf warten, daß ein so überwachter Joystick horizontal bewegt wird oder seine Taste gedrückt wird, so schreibt man in Keys eine 1 (Taste drücken), in Timeout eine 500 ($500/50=10$ Sekunden) und in XDelta eine 1.

Bevor man nun diese Überwachung starten kann, muß man natürlich einige Vorbereitungen treffen. Zuerst muß die Gameport-Device geöffnet werden, dann muß der Port, in dem der Joystick steckt, als Joystick-Port angemeldet und schließlich muß auf das gewählte Ereignis gewartet werden.

Die Kommandos dieser Device sind die folgenden:

Kommando	Name	Bedeutung
9	READEVENT	Überwachung starten
10	ASKCTYPE	Port-Typ abfragen
11	SETCTYPE	Port-Typ setzen
12	ASKTRIGGER	Auslösungsereignisse ermitteln
13	SETTRIGGER	Auslösungsereignisse setzen

Die möglichen Port-Typen, die mit SETCTYPE eingestellt werden können, sind:

Nummer	Name	Bedeutung
0	NOCONTROLLER	Abmelden des Ports
1	MOUSE	Mausport
2	RELJOYSTICK	Port für relative Joysticks
3	ABSJOYSTICK	Port für absolute Joysticks

zusätzlich gibt es noch den Typ

-1 ALLOCATED

welcher bei ASKCTYPE erscheinen kann; der Port ist dann bereits von einer anderen Task belegt.

Der Unterschied zwischen einem relativen und absoluten Joystick ist der, daß der X- bzw. Y-Wert eines relativen Joysticks beim Festhalten in einer Richtung ständig hoch- bzw. runtergezählt wird, beim absoluten Joystick gibt es nur eine Positionsveränderung pro Bewegung.

Um die Programmierung der Gameport-Device etwas zu demonstrieren, folgt nun ein Maschinenprogramm, das einen im rechten Port eingesetzten Joystick überwacht:

```
;***** Gameport-Device-Demo: Joystick 6/87 S.D. *****
```

```
ExecBase = 4  
FindTask = -294  
AddPort = -354  
RemPort = -360  
OpenLib = -408  
CloseLib = -414  
OpenDev = -444  
CloseDev = -450  
DoIo = -456  
SendIo = -462
```

run:

```

move.l execbase,a6      ;Zeiger auf EXEC-Bibliothek
sub.l  a1,a1            ;eigener Task
jsr    FindTask(a6)     ;Task suchen
move.l d0,readreply+$10 ;set SigTask

lea    readreply,a1
jsr    AddPort(a6)      ;Add Reply-Port

lea    devio,a1
move.l #1,d0            ;Unit 1: rechter Port
clr.l  d1
lea    devicename,a0
jsr    OpenDev(a6)      ;Gameport-Device öffnen
tst.l  d0
bne    error

```

*** Port-Typ setzen ***

```

move  #11,28(a1)        ;Command: SETCTYPE
move.l #Event,40(a1)    ;Puffer
move.l #1,36(a1)        ;Länge:
move.b #3,NextEvent     ;ABSJOYSTICK
lea    devio,a1
move.l #readreply,14(a1) ;set Reply-Port
move.l execbase,a6
jsr    DoIo(a6)         ;Joystick anmelden

```

*** Auslösung definieren ***

```

move  #13,28(a1)        ;Command: SETTRIGGER
move.l #trigger,40(a1)  ;Puffer
move.l #8,36(a1)        ;Länge

move  #3,Keys           ;DOWN & UP
move  #0,Timeout        ;Timeout
move  #1,XDelta         ;XDelta
move  #1,YDelta         ;YDelta

lea    devio,a1
move.l #readreply,14(a1) ;set Reply-Port
move.l execbase,a6
jsr    DoIo(a6)         ;Auslösung setzen

```

*** Überwachung starten ***

```

move    #9,28(a1)           ;Command: READEVENT
move.l  #Event,40(a1)       ;Puffer
move.l  #22,36(a1)          ;Länge: ein Event
clr.b   30(a1)              ;Flags

lea     devio,a1
move.l  #readreply,14(a1)   ;set Reply-Port
move.l  execbase,a6
jsr     DoIo(a6)            ;auf ein Event warten

```

*** Port wieder abmelden ***

```

move    #11,28(a1)          ;Command: SETCTYPE
move.l  #Event,40(a1)       ;Puffer
move.l  #1,36(a1)           ;Länge:
move.b  #0,NextEvent        ;NOCONTROLLER
lea     devio,a1
move.l  #readreply,14(a1)   ;set Reply-Port
move.l  execbase,a6
jsr     DoIo(a6)            ;Joystick abmelden

```

ende:

```

lea     readreply,a1
jsr     RemPort(a6)         ;Port entfernen

lea     devio,a1
jsr     closedev(a6)        ;Device schließen

```

error:

```

rts                      ;* Ende *

```

devicename: dc.b 'gameport.device',0

even

devio:

message: blk.b 20,0

io: blk.b 12,0

ioreq: blk.b 16,0

readreply: blk.l 8,0

Event:

NextEvent: dc.l 0
Class: dc.b 0
SubClass: dc.b 0
Code: dc.w 0
Qualifier: dc.w 0
ie_X: dc.w 0
ie_Y: dc.w 0
TimeStamp: dc.l 0,0

Trigger:

Keys: dc.w 0
Timeout: dc.w 0
XDelta: dc.w 0
YDelta: dc.w 0

Normalerweise ist es wohl sinnvoller, die Überwachung mit `SendIo()` zu starten, da das Programm dann nicht auf die Betätigung des Joysticks warten muß, sondern weiterarbeiten kann. Für dieses Beispiel ist es allerdings empfehlenswert zu warten, da sonst die Device vor dem Ereignis abgemeldet wird, was zu Problemen führen kann.

Anhang A: Bibliotheksfunktionen im Überblick

Die nun folgende Tabelle gibt Ihnen einen Überblick über alle verfügbaren Bibliotheken mit ihren Funktionen. Die Tabelle beginnt jeweils als Überschrift mit dem Namen der Bibliothek, in der die folgenden Funktionen liegen.

Diese Funktionen werden jeweils mit ihrem negativen Offset hex und dezimal, ihrem Namen und schließlich ihren Übergabeparameter dargestellt. Die Parameternamen stehen in Klammern hinter dem Funktionsnamen, die zweite Klammer enthält in gleicher Reihenfolge die Register, in denen diese Parameter übergeben werden müssen. Werden keine Parameter benötigt, wird dies durch () angedeutet.

clist.library

-\$001E	-30	InitCLPool (cLPool, size)(A0,D0)
-\$0024	-36	AllocCList (cLPool)(A1)
-\$002A	-42	FreeCList (cList)(A0)
-\$0030	-48	FlushCList (cList)(A0)
-\$0036	-54	SizeCList (cList)(A0)
-\$003C	-60	PutCLChar (cList,byte)(A0,D0)
-\$0042	-66	GetCLChar (cList)(A0)
-\$0048	-72	UnGetCLChar (cList,byte)(A0,D0)
-\$004E	-78	UnPutCLChar (cList)(A0)
-\$0054	-84	PutCLWord (cList,word)(A0,D0)
-\$005A	-90	GetCLWord (cList)(A0)
-\$0060	-96	UnGetCLWord (cList,word)(A0,D0)
-\$0066	-102	UnPutCLWord (cList)(A0)
-\$006C	-108	PutCLBuf (cList,buffer,length)(A0,A1,D1)
-\$0072	-114	GetCLBuf (cList,buffer,maxLength)(A0,A1,D1)
-\$0078	-120	MarkCList (cList,offset)(A0,D0)
-\$007E	-126	IncrCLMark (cList)(A0)
-\$0084	-132	PeekCLMark (cList)(A0)
-\$008A	-138	SplitCList (cList)(A0)
-\$0090	-144	CopyCList (cList)(A0)
-\$0096	-150	SubCList (cList,index,length)(A0,D0,D1)
-\$009C	-156	ConcatCList (sourceCList,destCList)(A0,A1)

console.library

- \$002A -42 CDInputHandler (events,device)(A0,A1)
 - \$0030 -48 RawKeyConvert (events,buffer,length,keyMap) (A0,A1,D1,A2)

diskfont.library

- \$001E -30 OpenDiskFont (textAttr)(A0)
 - \$0024 -36 AvailFonts (buffer,bufBytes,flags)(A0,D0,D1)

dos.library

- \$001E -30 Open (name,accessMode)(D1,D2)
 - \$0024 -36 Close (file)(D1)
 - \$002A -42 Read (file,buffer,length)(D1,D2,D3)
 - \$0030 -48 Write (file,buffer,length)(D1,D2,D3)
 - \$0036 -54 Input ()
 - \$003C -60 Output ()
 - \$0042 -66 Seek (file,position,offset)(D1,D2,D3)
 - \$0048 -72 DeleteFile (name)(D1)
 - \$004E -78 Rename (oldName,newName)(D1,D2)
 - \$0054 -84 Lock (name,type)(D1,D2)
 - \$005A -90 UnLock (lock)(D1)
 - \$0060 -96 DupLock (lock)(D1)
 - \$0066 -102 Examine (lock,fileInfoBlock)(D1,D2)
 - \$006C -108 ExNext (lock,fileInfoBlock)(D1,D2)
 - \$0072 -114 Info (lock,parameterBlock)(D1,D2)
 - \$0078 -120 CreateDir (name)(D1)
 - \$007E -126 CurrentDir (lock)(D1)
 - \$0084 -132 IoErr ()
 - \$008A -138 CreateProc (name,pri,segList,stackSize)(D1,D2,D3,D4)
 - \$0090 -144 Exit (returnCode)(D1)
 - \$0096 -150 LoadSeg (fileName)(D1)
 - \$009C -156 UnLoadSeg (segment)(D1)
 - \$00A2 -162 GetPacket (wait)(D1)
 - \$00A8 -168 QueuePacket (packet)(D1)
 - \$00AE -174 DeviceProc (name)(D1)
 - \$00B4 -180 SetComment (name,comment)(D1,D2)
 - \$00BA -186 SetProtection (name,mask)(D1,D2)
 - \$00C0 -192 DateStamp (date)(D1)
 - \$00C6 -198 Delay (timeout)(D1)
 - \$00CC -204 WaitForChar (file,timeout)(D1,D2)

-\$00D2 -210 ParentDir (lock)(D1)
 -\$00D8 -216 IsInteractive (file)(D1)
 -\$00DE -222 Execute (string,file,file)(D1,D2,D3)

exec.library

-\$001E -30 Supervisor ()
 -\$0024 -36 ExitIntr ()
 -\$002A -42 Schedule ()
 -\$0030 -48 Reschedule ()
 -\$0036 -54 Switch ()
 -\$003C -60 Dispatch ()
 -\$0042 -66 Exception ()
 -\$0048 -72 InitCode (startClass,version)(D0,D1)
 -\$004E -78 InitStruct (initTable,memory,size)(A1,A2,D0)
 -\$0054 -84 MakeLibrary (funcInit,structInit,libInit,dataSize,
 codeSize) (A0,A1,A2,D0,D1)
 -\$005A -90 MakeFunctions (target,functionArray,funcDispBase)
 (A0,A1,A2)
 -\$0060 -96 FindResident (name)(A1)
 -\$0066 -102 InitResident (resident,segList)(A1,D1)
 -\$006C -108 Alert (alertNum,parameters)(D7,A5)
 -\$0072 -114 Debug ()
 -\$0078 -120 Disable ()
 -\$007E -126 Enable ()
 -\$0084 -132 Forbid ()
 -\$008A -138 Permit ()
 -\$0090 -144 SetSR (newSR,mask)(D0,D1)
 -\$0096 -150 SuperState ()
 -\$009C -156 UserState (sysStack)(D0)
 -\$00A2 -162 SetIntVector (intNumber,interrupt)(D0,A1)
 -\$00A8 -168 AddIntServer (intNumber,interrupt)(D0,A1)
 -\$00AE -174 RemIntServer (intNumber,interrupt)(D0,A1)
 -\$00B4 -180 Cause (interrupt)(A1)
 -\$00BA -186 Allocate (freeList,byteSize)(A0,D0)
 -\$00C0 -192 Deallocate (freeList,memoryBlock,byteSize)(A0,A1,D0)
 -\$00C6 -198 AllocMem (byteSize,requirements)(D0,D1)
 -\$00CC -204 AllocAbs (byteSize,location)(D0,A1)
 -\$00D2 -210 FreeMem (memoryBlock,byteSize)(A1,D0)
 -\$00D8 -216 AvailMem (requirements)(D1)
 -\$00DE -222 AllocEntry (entry)(A0)
 -\$00E4 -228 FreeEntry (entry)(A0)
 -\$00EA -234 Insert (list,node,pred)(A0,A1,A2)
 -\$00F0 -240 AddHead (list,node)(A0,A1)
 -\$00F6 -246 AddTail (list,node)(A0,A1)

-\$00FC	-252	Remove (node)(A1)
-\$0102	-258	RemHead (list)(A0)
-\$0108	-264	RemTail (list)(A0)
-\$010E	-270	Enqueue (list,node)(A0,A1)
-\$0114	-276	FindName (list,name)(A0,A1)
-\$011A	-282	AddTask (task,initPC,finalPC)(A1,A2,A3)
-\$0120	-288	RemTask (task)(A1)
-\$0126	-294	FindTask (name)(A1)
-\$012C	-300	SetTaskPri (task,priority)(A1,D0)
-\$0132	-306	SetSignal (newSignals,signalSet)(D0,D1)
-\$0138	-312	SetExcept (newSignals,signalSet)(D0,D1)
-\$013E	-318	Wait (signalSet)(D0)
-\$0144	-324	Signal (task,signalSet)(A1,D0)
-\$014A	-330	AllocSignal (signalNum)(D0)
-\$0150	-336	FreeSignal (signalNum)(D0)
-\$0156	-342	AllocTrap (trapNum)(D0)
-\$015C	-348	FreeTrap (trapNum)(D0)
-\$0162	-354	AddPort (port)(A1)
-\$0168	-360	RemPort (port)(A1)
-\$016E	-366	PutMsg (port,message)(A0,A1)
-\$0174	-372	GetMsg (port)(A0)
-\$017A	-378	ReplyMsg (message)(A1)
-\$0180	-384	WaitPort (port)(A0)
-\$0186	-390	FindPort (name)(A1)
-\$018C	-396	AddLibrary (library)(A1)
-\$0192	-402	RemLibrary (library)(A1)
-\$0198	-408	OldOpenLibrary (libName)(A1)
-\$019E	-414	CloseLibrary (library)(A1)
-\$01A4	-420	SetFunction (library,funcOffset,funcEntry)(A1,A0,D0)
-\$01AA	-426	SumLibrary (library)(A1)
-\$01B0	-432	AddDevice (device)(A1)
-\$01B6	-438	RemDevice (device)(A1)
-\$01BC	-444	OpenDevice (devName,unit,ioRequest,flags)(A0,D0,A1,D1)
-\$01C2	-450	CloseDevice (ioRequest)(A1)
-\$01C8	-456	DoIO (ioRequest)(A1)
-\$01CE	-462	SendIO (ioRequest)(A1)
-\$01D4	-468	CheckIO (ioRequest)(A1)
-\$01DA	-474	WaitIO (ioRequest)(A1)
-\$01E0	-480	AbortIO (ioRequest)(A1)
-\$01E6	-486	AddResource (resource)(A1)
-\$01EC	-492	RemResource (resource)(A1)
-\$01F2	-498	OpenResource (resName,version)(A1,D0)
-\$01F8	-504	RawIOInit ()
-\$01FE	-510	RawMayGetChar ()
-\$0204	-516	RawPutChar (char)(D0)
-\$020A	-522	RawDoFmt ()(A0,A1,A2,A3)

-\$0210 -528 GetCC ()
 -\$0216 -534 TypeOfMem (address)(A1)
 -\$021C -540 Procedure (semaphore,bidMsg)(A0,A1)
 -\$0222 -546 Vacate (semaphore)(A0)
 -\$0228 -552 OpenLibrary (libName,version)(A1,D0)

graphics.library

-\$001E -30 BltBitMap (srcBitMap,srcX,srcY,destBitMap,destX,destY,
 sizeX,sizeY,minterm,mask,tempA) (A0,D0,D1,A1,D2,D3,D4,
 D5,D6,D7,A2)
 -\$0024 -36 BltTemplate (source,srcX,srcMod,destRastPort,destX,destY,
 sizeX,sizeY)(A0,D0,D1,A1,D2,D3,D4,D5)
 -\$002A -42 ClearEOL (rastPort)(A1)
 -\$0030 -48 ClearScreen (rastPort)(A1)
 -\$0036 -54 TextLength (RastPort,string,count)(A1,A0,D0)
 -\$003C -60 Text (RastPort,String,count)(A1,A0,D0)
 -\$0042 -66 SetFont (RastPortID,textFont)(A1,A0)
 -\$0048 -72 OpenFont (textAttr)(A0)
 -\$004E -78 CloseFont (textFont)(A1)
 -\$0054 -84 AskSoftStyle (rastPort)(A1)
 -\$005A -90 SetSoftStyle (rastPort,style,enable)(A1,D0,D1)
 -\$0060 -96 AddBob (bob,rastPort)(A0,A1)
 -\$0066 -102 AddVSprite (vSprite,rastPort)(A0,A1)
 -\$006C -108 DoCollision (rastPort)(A1)
 -\$0072 -114 DrawGList (rastPort,viewport)(A1,A0)
 -\$0078 -120 InitGels (dummyHead,dummyTail,GelsInfo)(A0,A1,A2)
 -\$007E -126 InitMasks (vSprite)(A0)
 -\$0084 -132 RemIBob (bob,rastPort,viewport)(A0,A1,A2)
 -\$008A -138 RemVSprite (vSprite)(A0)
 -\$0090 -144 SetCollision (type,routine,gelsInfo)(D0,A0,A1)
 -\$0096 -150 SortGList (rastPort)(A1)
 -\$009C -156 AddAnimObj (obj,animationKey,rastPort)(A0,A1,A2)
 -\$00A2 -162 Animate (animationKey,rastPort)(A0,A1)
 -\$00A8 -168 GetGBuffers (animationObj,rastPort,doubleBuffer)
 (A0,A1,D0)
 -\$00AE -174 InitGMasks (animationObj)(A0)
 -\$00B4 -180 GelsFuncE ()
 -\$00BA -186 GelsFuncF ()
 -\$00C0 -192 LoadRGB4 (viewport,colors,count)(A0,A1,D0)
 -\$00C6 -198 InitRastPort (rastPort)(A1)
 -\$00CC -204 InitVPort (viewport)(A0)
 -\$00D2 -210 MrgCop (view)(A1)
 -\$00D8 -216 MakeVPort (view,viewport)(A0,A1)
 -\$00DE -222 LoadView (view)(A1)

```

-$00E4 -228 WaitBlit ()
-$00EA -234 SetRast (rastPort,color)(A1,D0)
-$00F0 -240 Move (rastPort,x,y)(A1,D0,D1)
-$00F6 -246 Draw (rastPort,x,y)(A1,D0,D1)
-$00FC -252 AreaMove (rastPort,x,y)(A1,D0,D1)
-$0102 -258 AreaDraw (rastPort,x,y)(A1,D0,D1)
-$0108 -264 AreaEnd (rastPort)(A1)
-$010E -270 WaitTOF ()
-$0114 -276 QBlit (blit)(A1)
-$011A -282 InitArea (areaInfo,vectorTable,vectorTableSize)(A0,A1,D0)
-$0120 -288 SetRGB4 (viewPort,index,r,g,b)(A0,D0,D1,D2,D3)
-$0126 -294 QBSBlit (blit)(A1)
-$012C -300 BltClear (memory,size,flags)(A1,D0,D1)
-$0132 -306 RectFill (rastPort,xl,yl,xu,yu)(A1,D0,D1,D2,D3)
-$0138 -312 BltPattern (rastPort,rastPort,xl,yl,maxX,maxY,fillBytes)
      (A1,A0,D0,D1,D2,D3,D4)
-$013E -318 ReadPixel (rastPort,x,y)(A1,D0,D1)
-$0144 -324 WritePixel (rastPort,x,y)(A1,D0,D1)
-$014A -330 Flood (rastPort,mode,x,y)(A1,D2,D0,D1)
-$0150 -336 PolyDraw (rastPort,count,polyTable)(A1,D0,A0)
-$0156 -342 SetAPen (rastPort,pen)(A1,D0)
-$015C -348 SetBPen (rastPort,pen)(A1,D0)
-$0162 -354 SetDrMd (rastPort,drawMode)(A1,D0)
-$0168 -360 InitView (view)(A1)
-$016E -366 CBump (copperList)(A1)
-$0174 -372 CMove (copperList,destination,data)(A1,D0,D1)
-$017A -378 CWait (copperList,x,y)(A1,D0,D1)
-$0180 -384 VBeamPos ()
-$0186 -390 InitBitMap (bitMap,depth,width,height)(A0,D0,D1,D2)
-$018C -396 ScrollRaster (rastPort,dx,dy,minx,miny,maxx,maxy)(A1,D0,
      D1,D2,D3,D4,D5)
-$0192 -402 WaitBOVP (viewPort)(A0)
-$0198 -408 GetSprite (simpleSprite,num)(A0,D0)
-$019E -414 FreeSprite (num)(D0)
-$01A4 -420 ChangeSprite (vp,simpleSprite,data)(A0,A1,A2)
-$01AA -426 MoveSprite (viewPort,simpleSprite,x,y)(A0,A1,D0,D1)
-$01B0 -432 LockLayerRom (layer)(A5)
-$01B6 -438 UnlockLayerRom (layer)(A5)
-$01BC -444 SyncSBitMap (1)(A0)
-$01C2 -450 CopySBitMap (11,12)(A0,A1)
-$01C8 -456 OwnBlitter ()
-$01CE -462 DisownBlitter ()
-$01D4 -468 InitTmpRas (tmpras,buffer,size)(A0,A1,D0)
-$01DA -474 AskFont (rastPort,textAttr)(A1,A0)
-$01E0 -480 AddFont (textFont)(A1)
-$01E6 -486 RemFont (textFont)(A1)

```

```

-$01EC -492 AllocRaster (width,height)(D0,D1)
-$01F2 -498 FreeRaster (planePtr,width,height)(A0,D0,D1)
-$01F8 -504 AndRectRegion (rgn,rect)(A0,A1)
-$01FE -510 OrRectRegion (rgn,rect)(A0,A1)
-$0204 -516 NewRegion ()
-$020A -522 ** reserviert **
-$0210 -528 ClearRegion (rgn)(A0)
-$0216 -534 DisposeRegion (rgn)(A0)
-$021C -540 FreeVPortCopLists (viewPort)(A0)
-$0222 -546 FreeCopList (coplist)(A0)
-$0228 -552 ClipBlit (srcrp,srcX,srcY,destrp,destX,destY,sizeX,
sizeY,minterm)(A0,D0,D1,A1,D2,D3,D4,D5,D6)
-$022E -558 XorRectRegion (rgn,rect)(A0,A1)
-$0234 -564 FreeCprList (cprlist)(A0)
-$023A -570 GetColorMap (entries)(D0)
-$0240 -576 FreeColorMap (colormap)(A0)
-$0246 -582 GetRGB4 (colormap,entry)(A0,D0)
-$024C -588 ScrollVPort (vp)(A0)
-$0252 -594 UCopperListInit (copperlist,num)(A0,D0)
-$0258 -600 FreeGBuffers (animationObj,rastPort,
doubleBuffer)(A0,A1,D0)
-$025E -606 BltBitMapRastPort (srcbm,srcX,srcY,destrp,destX,destY,
sizeX,sizeY,minter)(A0,D0,D1,A1,D2,D3,D4,D5,D6)

```

icon.library

```

-$001E -30 GetWBOobject (name)(A0)
-$0024 -36 PutWBOobject (name,object)(A0,A1)
-$002A -42 GetIcon (name,icon,freelist)(A0,A1,A2)
-$0030 -48 PutIcon (name,icon)(A0,A1)
-$0036 -54 FreeFreeList (freelist)(A0)
-$003C -60 FreeWBOobject (WBOobject)(A0)
-$0042 -66 AllocWBOobject ()
-$0048 -72 AddFreeList (freelist,mem,size)(A0,A1,A2)
-$004E -78 GetDiskObject (name)(A0)
-$0054 -84 PutDiskObject (name,diskobj)(A0,A1)
-$005A -90 FreeDiskObj (diskobj)(A0)
-$0060 -96 FindToolType (toolTypeArray,typeName)(A0,A1)
-$0066 -102 MatchToolValue (typeString,value)(A0,A1)
-$006C -108 BumbRevision (newname,oldname)(A0,A1)

```

intuition.library

-\$001E	-30	OpenIntuition ()
-\$0024	-36	Intuition (ievent)(A0)
-\$002A	-42	AddGadget (AddPtr,Gadget,Position)(A0,A1,D0)
-\$0030	-48	ClearDMRequest (Window)(A0)
-\$0036	-54	ClearMenuStrip (Window)(A0)
-\$003C	-60	ClearPointer (Window)(A0)
-\$0042	-66	CloseScreen (Screen)(A0)
-\$0048	-72	CloseWindow (Window)(A0)
-\$004E	-78	CloseWorkBench ()
-\$0054	-84	CurrentTime (Seconds,Micros)(A0,A1)
-\$005A	-90	DisplayAlert (AlertNumber,String,Height)(D0,A0,D1)
-\$0060	-96	DisplayBeep (Screen)(A0)
-\$0066	-102	DoubleClick (sseconds,smicros,cseconds,cmicros) (D0,D1,D2,D3)
-\$006C	-108	DrawBorder (Rport,Border,LeftOffset,TopOffset) (A0,A1,D0,D1)
-\$0072	-114	DrawImage (RPort,Image,LeftOffset,TopOffset)(A0,A1,D0,D1)
-\$0078	-120	EndRequest (requester>window)(A0,A1)
-\$007E	-126	GetDefPrefs (preferences,size)(A0,D0)
-\$0084	-132	GetPrefs (preferences,size)(A0,D0)
-\$008A	-138	InitRequester (req)(A0)
-\$0090	-144	ItemAddress (MenuStrip,MenuNumber)(A0,D0)
-\$0096	-150	ModifyIDCMP (Window,Flags)(A0,D0)
-\$009C	-156	ModifyProp (Gadget,Ptr,Reg,Flags,HPos,VPos,HBody,VBody) (A0,A1,A2,D0,D1,D2,D3,D4)
-\$00A2	-162	MoveScreen (Screen,dx,dy)(A0,D0,D1)
-\$00A8	-168	MoveWindow (Window,dx,dy)(A0,D0,D1)
-\$00AE	-174	OffGadget (Gadget,Ptr,Req)(A0,A1,A2)
-\$00B4	-180	OffMenu (Window,MenuNumber)(A0,D0)
-\$00BA	-186	OnGadget (Gadget,Ptr,Req)(A0,A1,A2)
-\$00C0	-192	OnMenu (Window,MenuNumber)(A0,D0)
-\$00C6	-198	OpenScreen (OSArgs)(A0)
-\$00CC	-204	OpenWindow (OWArgs)(A0)
-\$00D2	-210	OpenWorkBench ()
-\$00D8	-216	PrintIText (rp,itext,left,top)(A0,A1,D0,D1)
-\$00DE	-222	RefreshGadgets (Gadgets,Ptr,Req)(A0,A1,A2)
-\$00E4	-228	RemoveGadgets (RemPtr,Gadget)(A0,A1)
-\$00EA	-234	ReportMouse (Window,Boolean)(A0,D0)
-\$00F0	-240	Request (Requester,Window)(A0,A1)
-\$00F6	-246	ScreenToBack (Screen)(A0)
-\$00FC	-252	ScreenToFront (Screen)(A0)
-\$0102	-258	SetDMRequest (Window,req)(A0,A1)
-\$0108	-264	SetMenuStrip (Window,Menu)(A0,A1)

-\$010E	-270	SetPointer (Window,Pointer,Height,Width,XOffset,YOffset) (A0,A1,D0,D1,D2,D3)
-\$0114	-276	SetWindowTitles (Window>windowTitle,screenTitle) (A0,A1,A2)
-\$011A	-282	ShowTitle (Screen>ShowIt)(A0,D0)
-\$0120	-288	SizeWindow (Window,dx,dy)(A0,D0,D1)
-\$0126	-294	ViewAddress ()
-\$012C	-300	ViewPortAddress (Window)(A0)
-\$0132	-306	WindowToBack (Window)(A0)
-\$0138	-312	WindowToFront (Window)(A0)
-\$013E	-318	WindowLimits (Window,minwidth,minheight,maxwidth, maxheight)(A0,D0,D1,D2,D3)
-\$0144	-324	SetPrefs (preferences,size,flag)(A0,D0,D1)
-\$014A	-330	IntuiTextLength (itext)(A0)
-\$0150	-336	WBenchToBack ()
-\$0156	-342	WBenchToFront ()
-\$015C	-348	AutoRequest (Window,Body,PText,NText,PFlag,NFlag,W,H) (A0,A1,A2,A3,D0,D1,D2,D3)
-\$0162	-354	BeginRefresh (Window)(A0)
-\$0168	-360	BuildSysRequest (Window,Body,PosText,NegText,Flags,W,H) (A0,A1,A2,A3,D0,D1,D2)
-\$016E	-366	EndRefresh (Window,Complete)(A0,D0)
-\$0174	-372	FreeSysRequest (Window)(A0)
-\$017A	-378	MakeScreen (Screen)(A0)
-\$0180	-384	RemakeDisplay ()
-\$0186	-390	RethinkDisplay ()
-\$018C	-396	AllocRemember (RememberKey,Size,Flags)(A0,D0,D1)
-\$0192	-402	AlohaWorkbench (wbport)(A0)
-\$0198	-408	FreeRemember (RememberKey,ReallyForget)(A0,D0)
-\$019E	-414	LockIBase (dontknow)(D0)
-\$01A4	-420	UnlockIBase (IBLock)(A0)

layers.library

-\$001E	-30	InitLayers (li)(A0)
-\$0024	-36	CreateUpfrontLayer (li,bm,x0,y0,x1,y1,flags,bm2)(A0,A1, D0,D1,D2,D3,D4,A2)
-\$002A	-42	CreateBehindLayer (li,bm,x0,y0,x1,y1,flags,bm2)(A0,A1,D0, D1,D2,D3,D4,A2)
-\$0030	-48	UpfrontLayer (li,layer)(A0,A1)
-\$0036	-54	BehindLayer (li,layer)(A0,A1)
-\$003C	-60	MoveLayer (li,layer,dx,dy)(A0,A1,D0,D1)
-\$0042	-66	SizeLayer (li,layer,dx,dy)(A0,A1,D0,D1)
-\$0048	-72	ScrollLayer (li,layer,dx,dy)(A0,A1,D0,D1)
-\$004E	-78	BeginUpdate (layer)(A0)

```

-$0054 -84 EndUpdate (layer)(A0)
-$005A -90 DeleteLayer (li,layer)(A0,A1)
-$0060 -96 LockLayer (li,layer)(A0,A1)
-$0066 -102 UnlockLayer (li,layer)(A0,A1)
-$006C -108 LockLayers (li)(A0)
-$0072 -114 UnlockLayers (li)(A0)
-$0078 -120 LockLayerInfo (li)(A0)
-$007E -126 SwapBitsRastPortClipRect (rp,cr)(A0,A1)
-$0084 -132 WhichLayer (li,x,y)(A0,D0,D1)
-$008A -138 UnlockLayerInfo (li)(A0)
-$0090 -144 NewLayerInfo ( )
-$0096 -150 DisposeLayerInfo (li)(A0)
-$009C -156 FattenLayerInfo (li)(A0)
-$00A2 -162 ThinLayerInfo (li)(A0)
-$00A8 -168 MoveLayerInFrontOf (layer_to_move,layer_to_be_in_front_
of)(A0,A1)

```

mathffp.library

```

-$001E -30 SPFix (float)(D0)
-$0024 -36 SPFlt (integer)(D0)
-$002A -42 SPCmp (leftFloat,rightFloat)(D1,D0)
-$0030 -48 SPTst (float)(D1)
-$0036 -54 SPAbs (float)(D0)
-$003C -60 SPNeg (float)(D0)
-$0042 -66 SPAdd (leftFloat,rightFloat)(D1,D0)
-$0048 -72 SPSub (leftFloat,rightFloat)(D1,D0)
-$004E -78 SPMul (leftFloat,rightFloat)(D1,D0)
-$0054 -84 SPDiv (leftFloat,rightFloat)(D1,D0)

```

mathieedoubbas.library

```

-$001E -30 IEEEEDPFix (integer,integer)(D0,D1)
-$0024 -36 IEEEEDPFlt (integer)(D0)
-$002A -42 IEEEEDPCmp (integer,integer,integer,integer)(D0,D1,D2,D3)
-$0030 -48 IEEEEDPTst (integer,integer)(D0,D1)
-$0036 -54 IEEEEDPAbs (integer,integer)(D0,D1)
-$003C -60 IEEEEDPNeg (integer,integer)(D0,D1)
-$0042 -66 IEEEEDPAdd (integer,integer,integer,integer)(D0,D1,D2,D3)
-$0048 -72 IEEEEDPSub (integer,integer,integer,integer)(D0,D1,D2,D3)
-$004E -78 IEEEEDPMul (integer,integer,integer,integer)(D0,D1,D2,D3)
-$0054 -84 IEEEEDPDIV (integer,integer,integer,integer)(D0,D1,D2,D3)

```

mathtrans.library

-\$001E	-30	SPAtan (float)(D0)
-\$0024	-36	SPSin (float)(D0)
-\$002A	-42	SPCos (float)(D0)
-\$0030	-48	SPTan (float)(D0)
-\$0036	-54	SPSincos (leftFloat,rightFloat)(D1,D0)
-\$003C	-60	SPSinh (float)(D0)
-\$0042	-66	SPCosh (float)(D0)
-\$0048	-72	SPTanh (float)(D0)
-\$004E	-78	SPExp (float)(D0)
-\$0054	-84	SPLog (float)(D0)
-\$005A	-90	SPPow (leftFloat,rightFloat)(D1,D0)
-\$0060	-96	SPSqrt (float)(D0)
-\$0066	-102	SPTieee (float)(D0)
-\$006C	-108	SPFieee (float)(D0)
-\$0072	-114	SPAsin (float)(D0)
-\$0078	-120	SPAcos (float)(D0)
-\$007E	-126	SPLog10 (float)(D0)

potgo.library

-\$0006	-6	AllocPotBits (bits)(D0)
-\$000C	-12	FreePotBits (bits)(D0)
-\$0012	-18	WritePotgo (word,mask)(D0,D1)

timer.library

-\$002A	-42	AddTime (dest,src)(A0,A1)
-\$0030	-48	SubTime (dest,src)(A0,A1)
-\$0036	-54	CmpTime (dest,src)(A0,A1)

translator.library

-\$001E	-30	Translate (inputString,inputLength,outputBuffer, bufferSize)(A0,D0,A1,D1)
---------	-----	--

Anhang B: Stichwortverzeichnis

*-Fenster	575
.info-Datei.....	559
512-KB-Erweiterungskarte	108
68000.....	13
8361.....	40
8362.....	40
8364.....	40
8367.....	40
8520.....	22
 Acknowledge	 74
Added.....	358
Adreßbus	15
Agnus	40, 46f
Aliasing Distortion	278
Allocate	414
AllocMem	404
AllocSignal	375
Amiga 1000	118
Amiga 500	61
Amiga-Betriebssystem	495
Analogeingänge	301
Ändern einer Library.....	347
AS	15
Ascending Mode	216
Assembler.....	318
Audio-Interrupts.....	272
Audio/Video-Schnittstellen	66
Ausgabeprozesse.....	417
Ausnahmezustände.....	391
Austastlücke.....	124
Autovektorielle Unterbrechung	20
 Basisadresse	 320
Baudrate	305, 310
Befehlsbyte.....	348
Betriebsparameter	314
Betriebssystem	317
Bewegen von Sprites	200
Bibliotheken	335, 619
Bibliotheksfunktionen	619
Bildschirmausgabe.....	124
Bildschirmfenster	166
Bitmap-Adressen	166

Bitmap-Block.....	532
Bitplanes	128, 153
Blitter	208
Blitter-Kontrollregister	224
Blitter-Operation	209ff, 244
Blitter-Steuerung	224
Blocks.....	520
Boolesche Algebra	218
Boolesche Gleichung.....	218
Boot-Block.....	523
Boot-Vorgang	522
Booten.....	522
Busy	74
Buszuweisung	20
Buszyklen.....	129
 C	318
C-Compiler	337
C-Strukturen	319
CD.....	77
Centronics-Schnittstelle.....	72f
Chip-Memory	492
Chip-RAM	63, 108
Chip-Register.....	120
CIA-Interrupts.....	443
CIA-Resource-Struktur	444
CIAs	109
CLI-Befehle.....	560
CLI-Kommando	563
Clist.library	619
Complex Interface Adapters (CIA).....	22
CON:-Fenster	567
Consol-Device	595
Console.library	620
Copper	140
Copper-Interrupt	147
Copperlist.....	141
Coprozessor	140
CSI	570
CTS.....	77
Cursortasten.....	572
Customchips.....	39f, 109
 Data-Block	531
Datei-Strukturen	523
Daten-Verteilung	523

Datenaustausch	564
Datenbus	15
Datenrichtungsregister	24
Datenteil	321
Datenübertragung	100
Datenumlenkung	501
DB	70
Deallocate	414
Denise	40, 52f
Descending Mode	216
Device-Handler	317
Device-Struktur	420
Devices	579
Dezibel	258
DG	70
DI	70
Direct Memory Access	121
Directory	509
Disc-Operating-System	495
Disk-Objekt	559
Diskcontroller	311
DISKDOCTOR	524
Diskette	520
Disketten-Operationen	503
Diskettenaufteilung	524
Diskettenlaufwerke	78
Diskfont.library	620
DMA	121
DMA-Controller	121
DMA-Kanäle	123
DMA-Kontrollregister	133
DMA-Vorgang	121
DOS	317
DOS-Bibliothek	495
DOS-Fehlermeldungen	516
DOS-Funktionen	498
Dos.library	496, 620
DR	70
Drucker	610
DSR	77
DTACK	15
DTR	77
Dual-Playfield-Modus	159, 183
DUMPRPORT	611

Early Read	110
Ein-/Ausgabe.....	564
Ein-/Ausgabe-Einheiten.....	121
Ein-/Ausgabe-Funktionen.....	498
Ein-/Ausgabekanäle	565
Eingabegeräte.....	303
Eingabeprozesse	417
Endmarkierung.....	352
Ereignis	573
Erstellen einer Library	348
Erzeugen neuer Tasks.....	367
Even Cycles.....	129
Exception	358
Excpetion Handler.....	391
Exec	317, 359
Exec.library.....	621
ExecBase-Struktur.....	458
Expansion-Port.....	90f
Extra-Halfbright	156
Farbpalette	154
Farbtabelle.....	154
Farbwahl	190
Fast RAM	44, 108
Fast-Memory	492
Fat-Agnus.....	61
FC0, FC1, FC2.....	19
Fehlermeldungen	516
FIFO-Warteschlange	386
File-Header-Block	528
File-List-Block	529
File-Tracer.....	589
Filehandle-Struktur	498
FileInfoBlock	507
FONT-Befehl	549
FreeMem	405
Füllen von Flächen.....	227
Fülloperation.....	231
Funktions-Offset	563
Funktionsaufruf.....	497
Funktionstasten	572
Gameport-Device	613
Gameports.....	87, 300
GamePortTrigger	614
Gary	63

Gemultiplexte Adressen	42
Genlock	70
Grafik	317
Grafikauflösungen	128
Graphics.library	623
Grundaufbau	41
Grundstrukturen	321
 HAM	 157
Hardware	12
Hash-Tabelle	526
Hertz	258
Hold-And-Modify	157
Horizontal Quadrature Pulse	295
Hüllkurve	279ff
Hunk	533
Hunk_break	538
Hunk_bss	535
Hunk_code	534
Hunk_data	534
Hunk_debug	537
Hunk_end	537
Hunk_ext	536
Hunk_header	537
Hunk_name	534
Hunk_overlay	538
Hunk_reloc16	536
Hunk_reloc32	535
Hunk_reloc8	536
Hunk_symbol	537
Hunk_unit	534
 I/O	 121
Icon	551
Icon-Library	555
Icon.library	625
InfoData	511
Inhaltsverzeichnis	508, 524
Initialisierung	323, 327
Initialisierungs-Tabelle	352
InputEvent	613
Insert	328
INT2, INT3, INT6	60
Interlace	125
Interlace-Modus	173
Interrupt-Ebenen	430

Interrupt-Funktionen	453
Interrupt-Handhabung	430
Interrupt-Kontrollregister (ICR)	33
Interrupt-Server.....	455
Interrupt-Struktur	432
Interrupts	137, 364, 431
Intuition	317
Intuition.library	626
IO-Steuerung	424
IO-Verwaltung.....	417
IPL0, IPL1, IPL2	19
Joystick	87, 294, 300, 613ff
Jump-Tabelle.....	317
Kickstart	117
Klangeffekte	273
Klangqualität	282
Kollision	203
Kommunikation zwischen Tasks.....	374
Kommunikationssystem.....	381
Kontrollregister	175
Lautstärke.....	273
Lautstärkemodulation	280
Lautstärkeverlauf	279
Layers.library	627
LDS	15
Library	335
Library-Funktionen	356
Library-Struktur.....	352
Lightpen.....	87, 135
List-Struktur	325
Liste	325, 333
Listen-Kopf.....	325
Listenverwaltung.....	328
Load-File	533
Lock	504
Long Frame.....	125
Maskierung.....	222
Mathffp.library	628
Mathieedoubbas.library	628
Mathtrans.library	629
Maus	294, 613
Mausport.....	87

Mehrstimmiger Klang	286
MemChunk-Struktur	412
MemHeader-Struktur	413
Memory mapped	23
Memory-List-Struktur	406
Message-Port	381
Message-Struktur	383
Message-System	381
Miniterms	218
Multitasking	317, 357
Musik	257
Narrator-Device	602
Node-Struktur	321
Node-Typ	323
NoFastMem-Routinen	492
Noten	283, 284
Obertöne	263
Obertonreihe	263
Object-File	533
Odd Cycles	129
Öffnen einer Library	341
Offset	325
Oktanten	233
OpenLibrary	496
OVL-Leitung	119
Paddles	87, 294, 301
PAL-Fernsehnorm	124
Paper Out	74
Parallel-Device	612
Parallelport	24
Parallelschnittstelle	74, 578
Parameterübergabe	497
Paula	40, 57f
Playfields	153, 161
Potentiometer	301
Potgo.library	629
Printer-Device	610
Prioritäten	200, 324, 362, 431
Program unit	533
Programm	547
Programmierung	318
Prozess-Verarbeitung	512
Prozessor Traps	392

Prozessor-Prioritäten	431
PRT:	610
Prüfsumme	353
Pseudo-Priorität	431
Public-Ports	382
Quantisierungsfehler	266, 282
R/W	15
Rasterstrahl	431
Rasterzeile	129
Rauschen	263
RAW-Input-Events	572
RAW-Keycodes	96
RAW:-Fenster	569
Ready	357
Refresh-Zyklen	136
Register	109, 319
Relocation	533
Removed	358
Reply-Port	387
Reset	17, 104
Reset-Routine	470
Resetfeste Module	480
Resetfeste Programme	470, 487
Resident-Strukturen	480
Resource-Struktur	447
RGB-Buchse	69f
ROM	117
Root-Block	525
RS232-Schnittstelle	76, 304f, 606
RTS	77
Rückgabeparameter	319
Running	357
RXD	77
Sample	265
Sampleperiod	270
Sampling-Rate	269
Schließen einer Library	343
Schnittstellen	65
Screen-Stuktur	341
Scrolling	168
Sektor	520
SER:	577
Serial-Device	606

Serielle Schnittstelle	75f, 304, 577
Serieller Port	32
Short Frame	125
Signalbits	375
Signalmaske	376
Smooth Scrolling	183
Soft-Interrupt	440
Sondercodes	102
Speicherbelegung	107
Speicherkapazität	402
Speicherverwaltung	401
Speicherzuweisung	410
Sprachausgabe	602
Sprite-Datenliste	192
Sprite-DMA	192
Sprite-Register	206
Sprites	153, 189
Sprungtabelle	335
Standard-I/O	565
Standard-Kommandos	581
Startup-Struktur	554
Steuerzeichen	569
Störfrequenzen	277
STROBE	74, 110
Stromversorgung	15, 94
Struktur einer Library	344
Supervisor-Modus	361
Synchronisation	101
 Takteingang	 15
Task	357
Task-Exceptions	391
Task-Funktionen	371
Task-Liste	359
Task-Scheduling	361
Task-Signale	375
Task-Stack	363
Task-Struktur	358
Task-Switching	360f, 364
Tastatur	96
Tastaturprozessor	97
Tiefpaßfilter	278
Timer.library	629
Tonausgabe	257
Track	520
Trackball	87

Trackdisk-Device	581
Translator.library	629
Trap-Befehle	396
Trap-Handler	393
Trap-Nummer	393
Traps	392f
Tremolo	259
TV-Mod-Buchse	67
TXD	77
UART	306
Übergabeparameter	318
UDS	15
User-Directory-Block	530
User-Mode	361
Verkettete Listen	321
Verkettungsteil	321
Verteilen der Rechenzeit	361
Vertical Quadrature Pulse	295
Vibrato	258
Videosignal	67
Waiting	357
Wellenform	261
WOM	118
XCLK	71
Zeitmultiplexverfahren	357

Schreiben Sie Ihre Programme in Maschinensprache – und Sie werden sehen, wie schnell ein Amiga sein kann. Das nötige Know-how liefert Ihnen dieses Buch: Grundlagen des 68000, das Amiga-Betriebssystem, Druckeransteuerung, Diskettenoperationen, Sprachausgabe, Windows, Screens, Register, Pull-Down-Menüs . . . Aber es wird auch gleich gezeigt, wie man mit den wichtigsten Assemblern arbeitet.



Dittrich
Amiga Maschinensprache
Hardcover, 282 Seiten,
DM 49,-
ISBN 3-89011-076-2

C an einem Wochenende? Durchaus möglich! Mit C für Einsteiger. Ein Einführungskurs, der Ihnen schnell und einfach die wichtigsten Grundlagen dieser Sprache vermittelt. Vom ersten Programm bis hin zu den Routinen in den Bibliotheken. Mit dem gesamten Sprachumfang und den besonderen Features von C. Zahlreiche Tips & Tricks zur Programmierung und eine Beschreibung der beiden Compiler Lattice C und Aztek runden das Ganze ab.



Schaun
Amiga C für Einsteiger
254 Seiten,
DM 39,-
ISBN 3-89011-107-6

DAS STEHT DRIN:

Von der Hardware über den Betriebssystemkern EXEC bis zum DOS finden Sie in diesem Buch die entscheidenden Informationen zum Amiga. Und zwar so verständlich, daß auch Nicht-Profis die Arbeitsweise des Amiga-Betriebssystems schnell verstehen werden.

Aus dem Inhalt:

- die Chips des Amiga (68000, CIA, Agnus, Denise, Paula)
- Die Schnittstellen (Video, Audio, RGB, Centronics, seriell, Floppy, Gameport, Expansionport, Tastatur)
- Programmierung der Hardware (Speicherbelegung, Interrupts, Copper, Blitter, Diskcontroller)
- Strukturen des EXEC (Node, List, Libraries, Tasks)
- Funktion des Multitasking (Task-switching, Kommunikation zwischen Tasks, Exceptions, Traps, Speicherverwaltung)
- I/O-Handhabung beim Amiga durch Devices und I/O-Request
- Interrupt-Handhabung und Verwaltung der Ressourcen
- RESET-feste Programme und Strukturen, Dokumentation der RESET-Routine
- EXEC-Base (Dokumentation und Nutzung der Systemvariablen)
- DOS-Bibliothek (Funktionen, Parameter, Fehlermeldungen)
- Disketten (Bootvorgang, Datenstrukturen, Programmaufbau, IFF-Format)
- Programmstart, Parameter, Aufruf von CLI und Workbench, detaillierte Beschreibung des internen Aufbaus der CLI-Befehle (interne DOS-Bibliothek)
- Ein-/Ausgaben (Tastatur, Bildschirm, Diskette, parallele und serielle Schnittstelle)
- Devices (Trackdisk, Console, Narrator, SER, PAR, PRT, Gameport)

UND GESCHRIEBEN HABEN DIESES BUCH:

Johannes Schemmel ist Hardware-Spezialist mit der Fähigkeit, Gesamtzusammenhänge verständlich darzustellen – wichtig für die Hardware-Beschreibung und das Zusammenwirken der einzelnen Komponenten mit der Software. Stefan Dittrich als 68000-Spezialist hat mit seinem ausgezeichneten Buch „Amiga Maschinensprache“ viel Aufmerksamkeit erzielt. Wer sich in irgendeiner Weise intensiv mit dem C64 beschäftigt, kennt Ralf Gelfands Titel „Floppybuch 1541“, „Intern“ oder „Das Anti-Cracker-Buch“.

ISBN N 3-89011-104-1 DM +069.00

DM 69,-
ÖS 538,-
sFr 67,-

**DATA
BECKER**



9 783890 111049

2701/8
DM 69.00
H. A. GRAFF
BUCHHANDLUNG



This was brought to you
from the archives of

<http://retro-commodore.eu>